ABSTRACT

GAO FEI. Controller in Core: An Adaptive Microarchitectural Model for System-level Optimization. (Under the direction of Professor Suleyman Sair).

Modern processors are utilizing more complex microarchitectures to extract more Instruction Level Parallelism (ILP) / Thread Level Parallelism (TLP) to improve performance. In addition to performance concerns, power and thermal issues have also become important for microprocessor designers because of the increasing power/heat density and resulting cooling costs. Meanwhile, many runtime architectural optimization approaches, called adaptive microarchitectures, are proposed to optimize system resources dynamically based on the characteristics of applications. However, most of them focus on improving a specific microarchitecture component or metric. In this dissertation, we argue that system-wide optimization is the future of adaptive microarchitectures that can balance the tradeoffs between different optimization choices to achieve maximum overall performance. We propose a runtime optimization architectural model - Controller in Core (CiC), which uses a dedicated element to synthesize and analyze the system-wide runtime information and make judicious optimization decisions. To demonstrate the CiC model, we present a performance-oriented adaptive microarchitecture - an adaptive value predictor that tailors its value prediction functionality based on runtime system performance bottleneck analysis. We propose an event counter based performance model that can accurately estimate the performance cost for critical system events. Based on this model, we propose the bottleneck vector as the basis of long-term performance bottleneck analysis and a runtime bottleneck phase tracking scheme. In addition, three bottleneck phase prediction schemes are studied. Based on performance bottleneck analysis, we develop adaptation algorithms to control

the adaptation of the adaptive value predictor. Our results show that the adaptive value predictor achieves 30% and 10% average performance gains when compared to the baseline and the traditional value predictor designs respectively.

Controller in Core: An Adaptive Microarchitectural Model for System-level Optimization

by

Fei Gao

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Electrical and Computer Engineering

Raleigh

2006

Approved By:

Dr. Vincent Freeh	Dr. Yan Solihin
Dr. Suleyman Sair Chair of Advisory Committee	Dr. Thomas M. Conte

To my daughter to be born

Biography

Fei Gao is born in Xi'an - the old capital of China. He graduated from University of Science and Technology of China (USTC) with M.S. and B.E. degree in Electronical Engineering in June 2001. Then he entered into Ph.D. program of ECE Dept. of North Carolina State University (NCSU). His research focuses on adaptive microarchitecture, data prefetching and branch prediction. Following graduation, he will work in Mathworks as senior software engineer engaging in code optimization.

Acknowledgements

I would like to thank all the members of my committee for their advice and help: My advisor - Dr. Suleyman Sair, Dr. Tom Conte, Dr. Yan Solihin and Dr. Vincent Freeh.

I would also like to thank my parents, sister and friends - without them, none of this would be possible.

Especially, I would thank my wife for her support on the way to my Ph.D. Degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few decades, general-purpose microprocessors have been evolving into million-transistor-on-chip systems from the first generation thousand-transistor microprocessors at the beginning of 1970's. The performance and complexity of microprocessors increase steadily along with hardware budgets, from 4 bits to 32 and 64 bits, from in-order single instruction execution cores to out-of-order superscalar and SMT (Simultaneous Multithreading) designs, and from a single core to CMPs (Chip Multiprocessors). As the number of on-chip transistors increases and the feature size decreases, power density in microprocessors has doubled almost every three years. Power dissipation has become a critical design constraint in high performance microprocessors. Because energy consumed by the microprocessor is converted into heat, the heat density is increased as well. The exponential rise in heat density is creating vast difficulties in reliability and high manufacturing cost. Generally, the design process is an optimization problem involving a number of dependent variables in a very large design space. These variables include system performance,

power consumption, hardware cost and architectural parameters. Exhaustive exploration of this design space is impractical. In practice, the common methodology is to limit the variable values based on past experiences to shrink the search space, and then to evaluate designs in this space using benchmark suites. In general, the parameter choices are made to achieve best *average performance*. The resulting design uses these fixed hardware resources, functionalities and parameters across all applications.

Yet, distinct applications have different characteristics. Their need for particular hardware resources, such as caches, issue queues, etc., vary significantly from application to application. For general-purpose microprocessors with fixed parameters yearning for good average performance, it comes as no surprise that a specific configuration may not be the optimal choice for certain applications. If an individual application's requirements do not match well with this particular configuration, the program may even exhibit poor performance. On the other hand, special-purpose processors can achieve significant performance gains on some specific applications, such as digital signal processing and graphics applications, because of their customized designs. In addition to the different characteristics of applications, several studies [13, 45] show that even a single application may exhibit significant variability during execution. The entire program execution consists of many portions with similar behavior, called *program phases*. Each phase may be quite different from others, while still having homogeneous behavior within a phase. Varying program phase behavior introduces another form of performance loss for fixed-parameter microprocessors.

Because of these inefficiencies in fixed-parameter general purpose microprocessor designs, microarchitects propose many runtime optimization schemes, called adaptive mi-

croarchitectures, to improve performance, power efficiency or thermal management. Unlike fixed-parameter designs, adaptive microarchitectures build flexibility into the chip, some design parameters can be reconfigured dynamically to meet application needs. With the increasing architectural complexity of a modern microprocessor, more architectural components become the target of runtime optimizations. Typical targets include instruction and data caches [1, 3, 29], pipeline width [2, 20] and issue queue [8, 19]. In addition, voltage and frequency scaling schemes have been shown to be very effective techniques for power and thermal management. For example, Dynamic Voltage Scaling (DVS) [37] switches supply voltage to low levels that results in lower execution frequency to reduce power consumption. Multiple Clock Domains (MCD) [42, 41] divides a chip into different clock domains that run at different frequency scaling regimes to save power. Recently, researchers propose several temperature aware microarchitectures and dynamic thermal management (DTM) [6, 34, 46] schemes to optimize thermal management.

Considering the many choices in reconfigurable microarchitectural components and runtime optimization targets, we argue that a dedicated hardware component to analyze the system-wide information to make optimization decisions is a worthy way to maximize the benefits from system resources. For example, if there are multiple on-chip components with power-saving modes, system-wide information can guide the system to the proper power-saving mode to minimize performance loss. In addition to guiding system-wide optimization, system-wide information analysis can benefit local optimizations as well. For example, a cache can turn part of a bank off to save power. Typically, those schemes are guided by local information, such as cache miss rate, to keep performance loss in an acceptable range.

If there is system-wide performance analysis that can estimate the performance impact of cache misses, a more efficient power saving strategy can be applied.

In this dissertation, we introduce a microarchitectural model, Controller-in-Core (CiC), to implement system-wide runtime optimization. We propose using a dedicated hardware component to analyze system-wide performance information to guide the adaptations of adaptive unit(s). Logically, the CiC adaptation procedure can be represented by a feedback control model. There are two main elements in this control system: the adaptive unit(s) and the control unit. The adaptive unit is the target of adaptive design, typically it is one of the hardware components or parameters of a processor. Unlike a fixed-parameter design, an adaptive unit is featured with extra control circuitry that enables reconfiguration during program execution. The key point of CiC is guiding runtime optimizations with the consideration of system performance and different optimization tradeoffs by using system-wide information instead of local information, to maximize the benefit of adaptive resources. There are two fundamental functions related to CiC design - distributed system information collecting and centralized information analysis. Corresponding to these two functions, there are two logical components inside CiC - a monitor and a controller. The monitor is in charge of collecting system-wide information and the controller is for information processing. CiC is a conceptual design model and its implementation depends on the specific microarchitecture and adaptation choices.

To demonstrate the CiC model, we present a performance-oriented adaptive microarchitecture design - an adaptive value predictor (AVP), which can adapt its functionality dynamically to performance bottlenecks. Traditional value predictors capture the repeating

patterns of register values and predict them to break data dependencies in early stages, instead of waiting for results to resolve, to improve ILP. A value predictor can also be used to help a branch predictor by predicting values and precomputing branch outcomes. And also, the value prediction table and computation logic can be used to produce memory address prediction for prefetching. The key idea of an adaptive value predictor is using a controller to analyze system runtime performance bottlenecks and make adaptation decisions to maximize the performance benefit of a value predictor. For this purpose, we propose a runtime performance bottleneck analysis model. Compared to the prior program phase work, our scheme knows where the performance limitations are. It identifies performance bottlenecks and applies optimizations accordingly, instead of searching for the optimal configuration like prior phase guided work does. We believe the CiC model can open the door for new adaptive microarchitectures and motivate microarchitects to produce more efficient adaptive designs.

The rest of this dissertation is organized as follows. We discuss the related work in Chapter 2. In Chapter 3, we describe the CiC concept in detail. Chapter 4 presents methodology of our experiments. In Chapter 5, we propose a runtime performance bottleneck analysis model. Chapter 6 presents a CiC application - adaptive value predictor. In Chapter 7, we present AVP monitor and AVP controller designs in detail. Chapter 8 describe AVP results. Finally, Chapter 9 concludes this dissertation and presents some future work.

Chapter 2

Related Work

This chapter summarizes the relevant prior work on adaptive microarchitectures and value prediction. We divide the past adaptive microarchitecture work into three categories - energy-efficient, performance-oriented and thermal-aware. They are discussed in the following sections respectively.

2.1 Energy-efficient adaptive microarchitectures

A number of proposals have been made for dynamically adapting the processor to achieve energy efficiency improvements.

Cache is the most active target of energy-efficient adaptivity because of its high percentage of total chip power consumption. Furthermore, caches have a regular structure and this makes adaptation control relatively easy from a circuit design perspective. Albonesi [1] proposes selective cache ways, which dynamically turns off cache ways to reduce dynamic energy dissipation. This technique exploits the subarray partitioning of set as-

sociative caches in order to disable ways of the cache during the period where full cache functionality is not required to reduce power dissipation with a small performance loss. Balasubramonian et al. [3] propose a single-level cache hierarchy, instead of the conventional multi-level design. This cache behaves as a *virtual two-level* cache hierarchy, and the sizes and associativities of these two levels are configurable. They dynamically detect the phase changes in an application by monitoring cache and TLB usage and reconfigure the cache hierarchy to reduce dynamic energy dissipation. Kaxiras et al. [29] reduce leakage power by invalidating and turning off the cache lines when they hold data that is not likely to be reused, based on the observation that cache lines are frequently used when first brought into the cache, and then have a period of dead time before they are evicted. Kim et al. [31] and Flautner et al. [18] propose a dynamic voltage scaling approach to implement a drowsy cache, where one can choose between two different supply voltages in each cache line. They observe that during a fixed period of time the activity in a cache is only centered on a small subset of the lines. They then exploit this behavior to reduce the leakage power of large caches by putting the cold cache lines into a state preserving, low-power drowsy mode. Moving lines into and out of drowsy state incurs a slight performance loss.

Similarly, the issue logic, as one of the most complex and power-consuming parts of a dynamically scheduled superscalar processor, is an important adaptive structure. Folegnani et al. [19] propose an issue queue scheme with the ability to dynamically resize the queue, based on the observation that mostly older instruction queue entries are used for issuing. They divide the entire instruction queue into 16 parts of eight entries each. Based on the committed IPC of the youngest part, the queue size can be reconfigured at run-time

to avoid the unnecessary search of parallel instructions in parts of the queue that hardly contribute to performance.

Buyuktosunoglu et al. [8] proposed an alternative circuit design for an adaptive issue queue. They divide the issue queue into separate chunks, connected via transmission gates. These gates are controlled by signals which determine whether a particular chunk is to be disabled to reduce the effective queue size. The queue size control signals are derived from counters that keep track of the active state of each entry on a cycle-by-cycle basis. After an execution interval, the decision to resize the queue can be made based on activity information.

In [2] Pipeline Balancing (PLB) is proposed. This scheme dynamically tunes the pipeline resources (issue width, function units) of a general purpose processor to the needs of the program by monitoring performance within each program. They analyze metrics (Issue IPC and Floating Point Issue IPC) for triggering PLB, and detailed instruction queue design, and observe the energy savings based on an extension of the Alpha 21264 processor. Soraya et al. [20] present an architectural mechanism that dynamically chooses between different processor configurations, in-order, out-of-order and pipeline-gating, that throttles instruction fetch to limit the speculative execution, by predicting IPC variation.

Dynamic voltage and frequency scaling of the CPU has been identified as one of the most effective ways to reduce energy consumption of a program. The total execution time (T) and energy consumption (E) of a program can be estimated by $T \approx W \cdot \frac{1}{f}$ and $E \approx C \cdot W \cdot V^2$ where W is the total number of execution cycles, f is the clock frequency, C is the effective switching capacitance, and V is the supply voltage. C and W are assumed to

be independent of frequency f. Since in dynamically voltage scaled (DVS) systems V varies approximately linearly with f ($V \propto f$), the performance-energy tradeoff of frequency scaling can be expressed as slowing down the clock frequency, which results in energy savings at the cost of decreased performance. Several manufacturers, such as Intel and Transmeta, have developed processors capable of global dynamic frequency and voltage scaling. Most current DVS related work [10, 25] has been focused on determining appropriate clock frequencies with respect to a predefined deadline for real-time systems. The basic idea is to recognize CPU slack time and then scale voltage and frequency to save power while meeting the deadline.

As clock frequency increases and feature size decreases, clock distribution and wire delays present a growing challenge to the designers of singly-clocked, globally synchronous systems. Semeraro et al. [42] propose a Multiple Clock Domain (MCD) processor, in which the chip is divided into several clock domains, within each of which independent voltage and frequency scaling can be performed. In their work, they use four clock domains, corresponding to the front end (including L1 instruction cache), integer units, floating point units, and load-store units (including L1 data cache and L2 cache). Their design attains 20% energy savings with less than 4% performance loss.

2.2 Temperature-aware microarchitecture

The increase in chip power density leads to higher chip temperature, which has a significant impact on the overall energy consumption, product reliability, and platform cost. With aggressive transistor scaling, the transistor's subthreshold leakage (static power)

is becoming a significant part of total power consumption, and this leakage is exponentially related to temperature. Therefore, a small increase in average temperature may result in significant energy loss for deep-submicron designs. In addition, device reliability degrades quickly with temperature. And finally, the higher the temperature, the more expensive the thermal solutions, and this significantly affects the platform costs.

Brooks et al. [6] propose an Icache-toggling technique, which disables instruction fetch to control the amount of in-flight instructions, thus reducing on-chip activity and eventually reducing temperature. Icache-toggling disables the instruction fetch unit for a specified interval. The length of this interval depends on power saving needs. Lim et al. [34] propose a two-pipeline microarchitecture, a core pipeline and a simple pipeline. The core pipeline is a general superscalar out-of-order pipeline and the simple one is an in-order scalar pipeline. During normal operation, only the primary core is operating, and thermal sensors distributed across the die are used to detect temperature excursions. When temperature exceeds a given threshold, core pipeline will be clock-gated and the simple pipeline will be engaged to relieve temperature. Skadron et al. [46] propose an architecture-level thermal model - Hotspot, based on thermal resistances and capacitances model. In addition, they also introduce several thermal management techniques, such as temperature tracking frequency scaling and migrating computation to spare hardware units.

2.3 Performance-oriented adaptive microarchitectures

Another category of adaptive microarchitectures is performance-oriented adaptive microarchitectures. They dynamically adapt system resources to program needs to improve

system performance, based on the observed values of predetermined metrics, such as IPC, or cache miss rate and so on.

Since accessing the memory is still a performance bottleneck in current microprocessors, caches get a lot of attention in performance-oriented adaptive microarchitectures as well. Veidenbaum [53] uses an adaptive strategy to adjust cache line size dynamically to an application. Qureshi et al. [38] propose a mechanism to provide the ability to perform global replacement while maintaining the constant hit latency of a set-associative cache, called the Variable-Way Set Associative Cache, or simply V-way cache. For the V-way cache, they use a global replacement policy based on an access frequency measure called Reuse Replacement. Reuse Replacement performs comparably to a least recently used (LRU) policy at a fraction of the hardware cost and complexity. Other studies reduce the number of cache misses by increasing the effectiveness of the memory hierarchy.

Johnson et al. [26] present a technique for dynamic analysis of program data access behavior, which is then used to proactively guide the placement of data within the cache hierarchy in a location-sensitive manner. They introduce the concept of a macroblock, which allows to feasibly characterize the memory locations accessed by a program, and a Memory Address Table, which performs the dynamic reference analysis. Speight et al. [48] propose simple architectural extensions and adaptive policies for managing the L2 and L3 cache hierarchy in a CMP system.

Another example of performance-oriented adaptive designs is branch history length adaptation. Accurate branch prediction is essential for obtaining high performance in pipelined superscalar processors that execute instructions speculatively. Some predictors

use global branch history to exploit the correlations between branches. However, the best global branch history length depends on the code, the input data and frequency of context switches. Juan et al. [28] explore dynamic history-length fitting (DHLF) and develop a method for dynamically selecting a history length that accommodates the current workload.

Balasubramonian et al. [4] dynamically allocate available registers between the primary program thread and a future thread. The future thread executes instructions when the primary thread is limited by resource availability. The future thread is not constrained by in-order commit requirements. It is therefore able to examine a much larger instruction window and jump far ahead to execute ready instructions. Results are communicated back to the primary thread by warming up the register file, instruction cache, data cache, and instruction reuse buffer, and by resolving branch mispredictions early. Cazorla et al. [9] introduce the concept of dynamic resource control in SMT processors. Using this concept, they propose a novel resource allocation policy for SMT processors. This policy directly monitors the usage of resources by each thread and guarantees that all threads get their fair share of the critical shared resources, avoiding monopolization. They also define a mechanism to allow a thread to borrow resources from another thread if that thread does not require them, thereby reducing resource induced stalls.

2.4 Performance modeling

In our work, we propose an event counter based performance model in order to identify the runtime performance bottlenecks. This section discusses several methods related to performance modeling.

Software-based analysis tools are very useful in obtaining the basic information about a program's execution. Simulators and instrumentation tools are widely used for this purpose. PROTES [5], a MIMD architecture simulator, can simulate parallel machines to provide accurate information about the timing and behavior of an application. CPROF [33], a cache profiler, can identify cache performance bottlenecks. ATOM [51] implements a very flexible and efficient code instrumentation tool based on OM link-time technology. The instrumented performance analysis routines are linked with the original program with a procedure call interface.

Meanwhile, modern microprocessors provide hardware support for performance analysis, called event counters. These counters are built into the chip to measure corresponding events. In MIPS R10000 [55], there are two event counters that can capture 30 system events. The Pentium 4 [50] processor provides 18 event counters and 48 event detectors. These event counter based performance analysis tools provide an accurate and valuable inside view of dynamic program behavior. However, event counters can not accurately attribute those events to instructions, especially for out-of-order machines. For this reason, Dean et al. [12] proposed an instruction-level profiling technique - ProfileMe that samples instructions. As a sampled instruction moves through the processor pipeline, a detailed record of all interesting events and pipeline stage latencies is collected. ProfileMe also supports paired sampling, which captures information about the interactions between concurrent instructions. Similarly, to reveal the interactions of different instructions, Fields et al. [17] proposed an interaction cost model that can associate execution cycles to one instruction, or multiple instructions processed in parallel.

In our work, we focus on long-term performance behavior rather than fine-grain analysis. In addition, we propose a low-overhead performance bottleneck model based on event counters instead of expensive instruction-level statistics.

2.5 Long-term program behavior

In our work, we study the program performance bottleneck behavior from a long-term viewpoint, based on our performance bottleneck analysis model. There is a plethora of relevant work on long-term program behavior. In general, these techniques try to identify and predict "program phases" (the execution portions that have similar behavior) by tracking various program signatures.

Balasubramonian et al. [3] use a conditional branch counter to detect program phase changes. The counter keeps track of the number of dynamic conditional branches executed over a fixed execution interval (measured in terms of the dynamic instruction count). Phase changes are detected when the difference in branch counts of consecutive intervals exceeds a threshold. Their scheme does not use a fixed threshold. Rather, the detection algorithm dynamically varies the threshold throughout the execution of the program based on the cache miss rate and the branch frequency.

Dhodapkar et al. [13] define a program phase as the instruction working set of the program (i.e. the set of instructions touched in a fixed interval of time). Program phase changes are detected by comparing consecutive instruction working sets using a similarity metric called the relative working set distance. Because complete working sets can be too large to efficiently represent and compare in hardware, they propose the use of compressed

representations of working sets called working set signatures. Signatures are compared using a metric called the relative signature distance. Phase changes are detected when the relative signature distance between consecutive intervals exceeds a preset (fixed) threshold. They show that signatures as small as 32 bytes in size can be used to resolve program phases in most benchmarks studied.

Sherwood et al. [45, 44] propose the use of basic block vectors (BBVs) to detect program phase changes. BBVs keep track of execution frequencies of basic blocks touched in a particular execution interval. Phase changes are detected when the Manhattan distance between consecutive BBVs exceeds a preset threshold. Because entire BBVs cannot be stored in hardware, BBVs are approximated by hashing into an accumulator table containing a few counters. Their results indicate that as few as 32 24-bit counters are sufficient to represent BBVs in hardware.

Dhodapkar et al. [14] compare these three dynamic program phase detection techniques: using instruction working sets, BBVs, and conditional branch counts. They find BBV techniques perform better than the other techniques providing higher sensitivity and more stable phases. Lau et al. [32] examine different program structures that are independent from underlying architecture performance metrics for capturing phase behavior. The structures include basic blocks, loop branches, procedures, opcodes, register usage and memory address information. They compare these different structures in terms of their ability to create homogeneous phases, and evaluate the accuracy of using these structures to pick simulation points.

Shen et al. [43] propose a mechanism to predict data locality phases. At first,

they detect the locality phase by examining the data reuse distances. Then they analyze the instruction trace and identify the phase boundaries in the code. Finally, after using grammar compression to identify phase hierarchies, they insert program markers through binary rewriting. During execution, the program uses the first few instances of a phase to predict all its later executions. The new analysis considers both program code and data accesses.

The key difference between our work and previous long-term program behavior analysis is that we focus on performance bottleneck analysis - the cause of performance behavior - to provide system-wide performance diagnosis from a long-term viewpoint. This helps our technique identify beneficial configurations without searching the adaptive design space.

2.6 Value prediction

To demonstrate the CiC model, we propose an adaptive value predictor, which uses one value prediction table to implement four functionalities based on prior value prediction work. These functions are traditional value prediction, branch prediction through value prediction and prefetching. We present these work in this section.

Traditional value predictors exploit "value locality" to predict register values in order to break up data dependence chain and improve ILP. There are three typical value stream patterns - last value, stride, and context-based. The last value predictor [36, 35] uses the current value of an instruction as its next value prediction. The stride value predictor [11] keeps track of not only the last value brought in by an instruction, but also the difference

between that value and the previous value. The difference between these values is called the stride. The new value prediction is the sum of the last value and the stride. Two-Delta stride value predictor is verified to be more accurate. It only replaces the predicted stride with a new stride if the new stride has been observed twice. Each entry of the prediction table contains a tag, the predicted value, the predicted stride, and the last stride. The context predictor [39, 40] bases its prediction on the last several values seen. A table called the Value History Table (VHT) contains the last N values seen for each entry. Another table, called the Value Prediction Table (VPT), contains the actual value to be predicted. An instruction's PC is used to index into the VHT, which holds the past history of the instruction. The history values in this entry are hashed into a single index into the VPT. This entry in the VPT contains the value to be predicted. The context predictor is able to keep track of a finite number of reference patterns that are not necessarily constrained by a fixed stride. Besides the stream pattern difference, there is also value prediction classification based on where the pattern is captured, called `local` and `global` [56] value prediction. For local value prediction, the value stream is observed for each instruction. For global value prediction, the value stream is observed for all dynamic instructions according to their execution order.

Gonzalez [22] proposed a branch prediction scheme - Branch Predictor through Value Prediction (BPVP). This branch predictor predicts the outcomes of branches by predicting the values of their inputs and performing an early computation of their results according to the predicted values. A chooser chooses the final prediction from a general prediction and the precomputed prediction.

Data prefetching is an effective technique to tolerate memory latency. Research shows that the memory address stream is highly predictable. There are two basic stream patterns: stride and context. Corresponding prefetchers are proposed to capture those patterns. A *stride* predictor [11, 16] keeps track of not only the last address referenced by a load, but also the difference between the last address of the load and the address before that. The predictor speculates that the new address seen by the load will be the sum of the last address value and the stride. In [16, 39], two-delta stride prefetcher is proposed, which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row. *Context* [39, 40, 54] and *Markov* [27] predictors are fundamentally similar, in that each predictor bases its prediction on a set of past values seen. An order k context/Markov predictor uses the k past values to predict the next one. While making a prediction, the prediction table is also updated to record transitions in the address stream dynamically.

Individual instruction cache misses are usually more expensive than data cache misses as instruction misses stall the processor pipeline, while a significant proportion of data misses can often be overlapped with other instructions in an out-of-order execution engine. Many previous work [23, 24, 47, 49] about instruction prefetching have been proposed. Typically, instruction prefetching can be divided into two basic categories: sequential instruction prefetching and non-sequential instruction prefetching. The simplest sequential instruction prefetching is next-line prefetching. The prefetcher fetches next cache line when there is an instruction cache miss. For next-line prefetching, there is also a next-N-line derivation. For this scheme, the prefetcher fetches N next lines instead of just one. Besides sequential instruction misses, control flow changes result in instruction misses as well.

There are some schemes to target those non-sequential misses. For example, Hsu et al. [24] proposed "target" prefetching. They used a table to remember the sequence of cache lines fetched. This table is searched by the current fetch address. If there is a hit, the corresponding next address is prefetched.

Chapter 3

Controller in Core

This chapter presents the CiC design model. Since CiC is a conceptual microarchitectural model for adaptive microarchitectures, we discuss it mostly in a logical and conceptual scope in this chapter. Its physical design depends on the specific microprocessor and applications.

3.1 CiC concept

Modern microprocessors have become more complex and feature more and bigger hardware components to extract more ILP/TLP. In addition to performance, power consumption becomes a critical design constraint as the power density increases. The resulting heat density is also increasing correspondingly. Adaptive microarchitectures try to meet the application needs with built-in flexibility. Adaptation can target practically any component of the microprocessor, and the adaptation goals include performance, power consumption and heat dissipation. We argue that future adaptive microprocessors will feature more

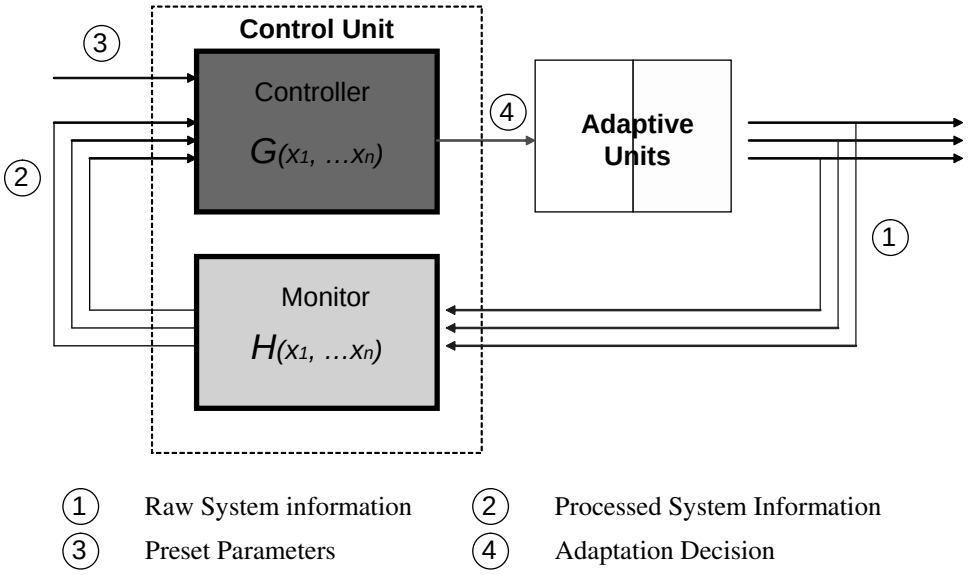

Figure 3.1: CiC feedback control model

adaptive targets on the chip. The resulting question is how to effectively manage these adaptation possibilities to maximize the resource usage. We propose the CiC design model to implement the system-wide analysis as an answer to this question.

The key point of CiC is guiding runtime optimizations with the consideration of overall system performance and different optimization tradeoffs by using system-wide information instead of local information. There are two fundamental functions related to CiC design - distributed system information collecting and centralized information analysis. Corresponding to these two functions, there are two logic components inside CiC - a monitor and a controller. The monitor is in charge of collecting system-wide information and the controller is for information processing.

3.2 CiC logical feedback control model

The CiC logic model is shown in Figure 3.1. Logically, the CiC adaptation procedure can be represented by a feedback control model. There are two main elements in this control system: the adaptive unit(s) and the control unit. The adaptive unit is the target of the adaptive design, typically it is one of the hardware components or parameters of a processor. Unlike a fixed-parameter design, an adaptive unit features extra control circuitry that enables reconfiguration during program execution. Typical examples of adaptive units are as follows.

- *caches and TLBs* - line size or associativity is adjusted to accommodate distinct memory access patterns.
- *memory hierarchy* - cache resources are divided among levels of the memory hierarchy.
- *branch predictor* - the length of global history register can be varied during execution.
- *issue window* - parts of the issue window is disabled for power efficiency when the entire issue window is not needed.
- *pipeline* - a portion of the pipeline resources can be turned off to save power or pipeline can switch between in order, out-of-order, and pipeline gating modes of operation.

The control unit is the core of the CiC model. It is in charge of making adaptation decisions. As Figure 3.1 shows, it consists of a monitor and a controller. The monitor is on the path of the feedback loop that transmits the runtime system information back to the controller. The controller is directly connected to the adaptive units to guide their adaptation. The

input of the monitor is raw system information that is related to the adaptation schemes. The output of the monitor is the processed system information, because the raw information can not be used for adaptation decisions directly. Usually, information processing is based on a specific analytical model. The input of the controller consists of two parts - one is the processed system information from the output of the monitor. The other is some preset built-in parameters, such as threshold values that come from empirical experiments. The output of the controller is the adaptation decision that changes the configuration of the adaptive unit at runtime.

The CiC operation cycle is as follows.

- The monitor collects related system information across the chip. Typically, data is collected at a fixed sampling period.
- The monitor processes the collected information, feeds the processed information back into the controller.
- The controller analyzes the information feed and makes an adaptation decision based on an adaptation algorithm.
- The adaptive unit tunes its configuration according to the controller's decision.
- The reconfigured adaptive unit changes the related system information. And then a new CiC operation cycle starts. The monitor will feed the changed system information back into the control loop to show the effectiveness of the previous adaptation decision and continually adjust adaptation decision based on performance change.

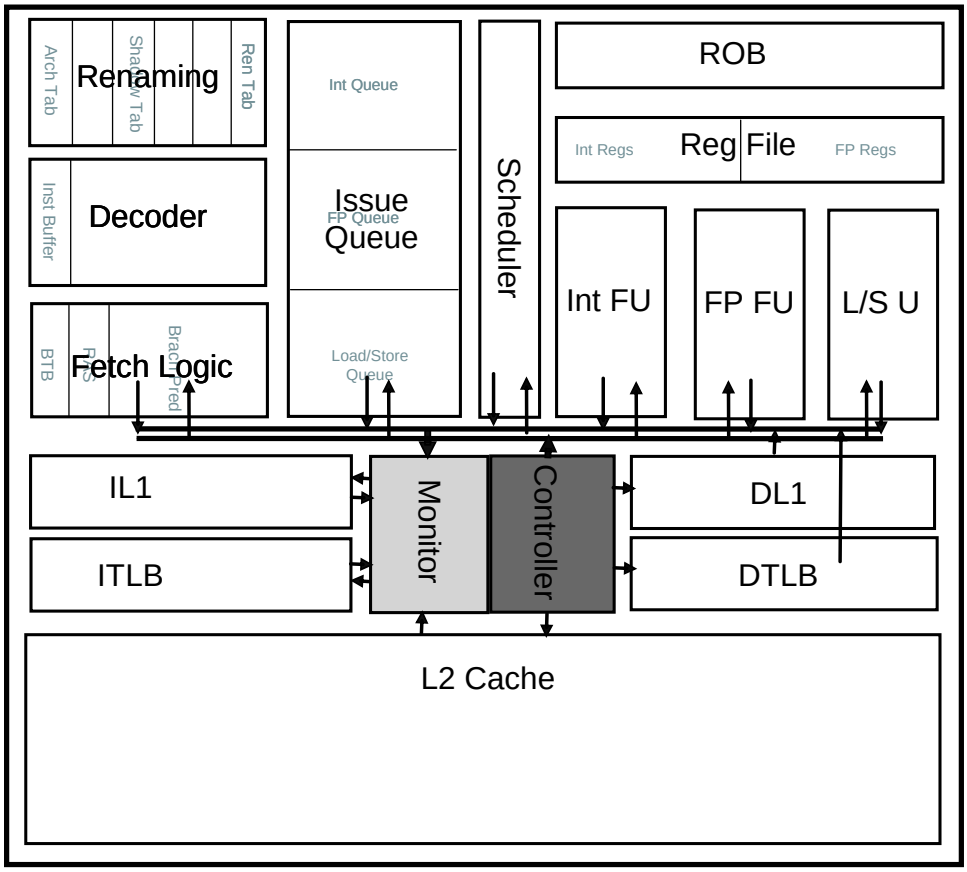

Figure 3.2: An example CiC floorplan

3.3 CiC chip floorplan

An example implementation of CiC chip floorplan is shown in Figure 3.2. It is a hardware implementation of the CiC logic model. There is a dedicated onchip hardware component - control unit that monitors and controls system optimizations. The monitor implementation is similar to an event counter design, where an event detector resides in each target component. When a corresponding event occurs, a signal is created to increment the counter. For a sampling period, the event counters is read by the monitor and reset. And then the monitor processes the row value from these event counter to create bottleneck

vector and feed it into the controller. The controller is an application specific module to implement a specific adaptation algorithm based on the information from the monitor.

Chapter 4

Methodology

4.1 Simulator

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [7], a suite of functional and timing simulation tools for the Alpha AXP ISA. Table 4.1 presents the configuration parameters for the baseline microarchitecture.

4.2 Benchmarks

This work studies 12 SPEC 2000 integer benchmarks. They are `mcf`, `parser`, `vpr`, `gzip`, `crafty`, `gcc`, `gap`, `parser`, `perl`, `eon` and `twolf`. Each program was simulated for 10 Billion committed instructions from the beginning. The benchmark details are presented in Table 4.2.

Table 4.1: Architectural configurations

Fetch/Retire width	4 instructions
Branch predictor	gshare 16K entries
BTB	1K entries
RUU size	128
Load/store queue	64
Functional units	4 intALU, 2 int mul/div
I-TLB	64 entries, 30 cycle miss latency
D-TLB	128 entries, 30 cycle miss latency
I-cache L1	32KB, 2-way set associative, 32 byte line, 1 Cycle hit latency
D-cache L1	32KB, 2-way set associative, 32 byte line, 1 Cycle hit latency
L2 cache	1MB, 4-way set associative, 64 byte line, 12 Cycle hit latency, 120 Cycle miss latency

Table 4.2: The benchmarks studied in this dissertation

benchmark	input	description
gzip	input.source	Compression
vpr	*.in	FPGA circuit placement and routing
gcc	166.i	C programming language compiler
mcf	inp.in	Combinatorial optimization
crafty	crafty.in	Chess game playing
parser	ref.in	Word processing
eon	chair	Computer visualization
perl	splitmail	PERL programming language
gap	ref.in	Group theory, interpreter
vortex	lendian	Object-Oriented database
bzip2	input.source	Compression
twolf	ref	Place and route simulator

Chapter 5

Performance Bottleneck Modeling

The function of CiC monitor is collecting system-wide performance information for performance bottleneck analysis. This chapter presents the performance bottleneck model in detail.

5.1 Performance modeling

The performance of a modern microprocessor benefits from improvements in both circuit technology and microarchitecture. Over the past decade, a typical microarchitecture has evolved from in-order and scalar to out-of-order and superscalar with speculative execution. However, as processors enjoy high instruction level parallelism from modern microarchitectures, system-level performance becomes harder to understand and the question, "Where have the cycles gone ?", becomes harder to answer, because many events overlap and interact with each other. An effective performance analysis becomes more essential for microarchitects and application developers to diagnose the code behavior and provide

optimizations.

There are many approaches developed to analyze program performance. The most convenient way is software-based analysis tools, such as simulation [21, 5, 33] and instrumentation [51]. Simulation can provide details of program behavior, but causes several orders of magnitude slowdown. Instrumenting original code can catch dynamic events, but there still is code overhead and also the instrumentation code may change the behavior of the original program. On the hardware side, modern microprocessors started providing hardware support for performance analysis, called event counters [55, 50]. These counters give an inside view of how the program interacts with the underlying hardware. Users can access the counters with operating system and library support.

Modern microprocessors are able to execute a program at a speed of millions of instructions per second. As a result, long-term program behavior interests researchers because of the potential long-term optimizations, such as power and thermal management. It has been shown that programs exhibit periods of similar behavior during their execution, called program phases [3, 13, 45]. The reason of the phase phenomena is the regularity of code execution. Tracking the footprint of instructions [45] and working sets [13] are typical approaches to capture program phases.

Performance bottlenecks are both hardware and software dependent. Hardware limitations, such as limited cache capacity, can slowdown program execution. The characteristics of a program impacts performance as well, such as the amount of available inherent code parallelism. Most previous performance bottleneck analysis are either profiling based offline or fine-grain instruction level dynamic schemes with the consideration of parallel

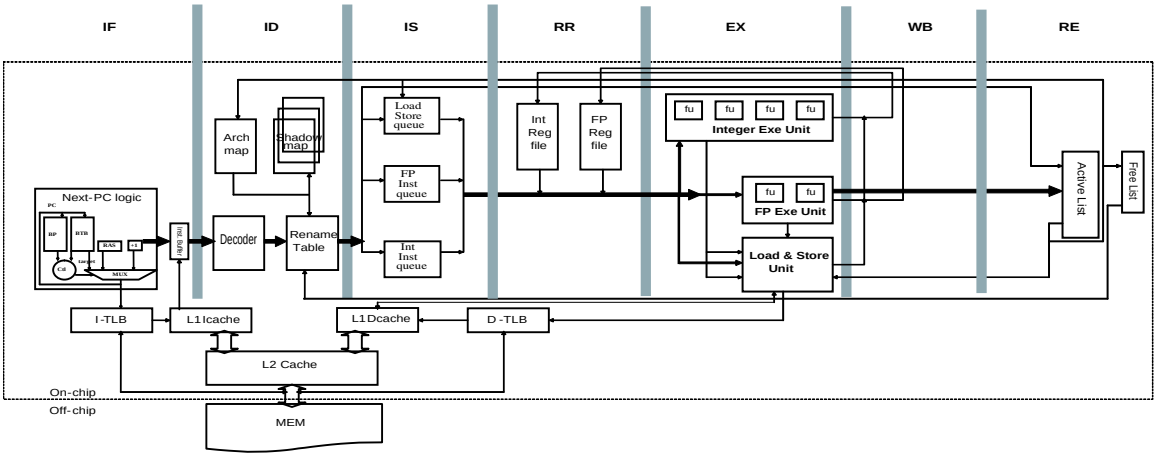

Figure 5.1: 7 stage out-of-order superscalar microarchitecture

instruction execution. In this paper, we analyze performance bottlenecks from a long-term point of view. Unlike previous long-term work that catch the regular patterns of program behavior to predict performance changes, our work provides a system-wide performance diagnosis to find out what causes the current performance change and what the next performance bottleneck will be.

In our work, we propose

- A counter-based long-term performance model to quantify the performance impact of different system events,
- A bottleneck vector based bottleneck phase tracking scheme to capture program bottleneck behavior at runtime,
- Performance bottleneck phase prediction schemes to guide system optimization.

5.2 Performance bottlenecks

Performance becomes hard to understand in a modern superscalar out-of-order microprocessor because of the increasing number of inflight instructions and the interactions between them. Therefore, it is necessary to review the instruction flow of a modern microarchitecture and the potential factors that could cause performance slowdowns, before we do further performance bottleneck analysis.

5.2.1 Superscalar out-of-order microarchitecture

Figure 5.1 shows a 7-stage out-of-order microarchitecture. The stages are IF (Instruction Fetch), ID (Instruction Dispatch), IS (Issue), RR (Register Read), EX (Execution), WB (Writeback) and RE (Retire).

In the IF stage, the next instruction logic produces the next PC the processor will fetch. The BTB (Branch Target Buffer) identifies the type of the current instruction. For conditional branches, the branch predictor is involved for making a Taken/NotTaken prediction. Unconditional branches are always predicted taken and the next PC is obtained from this BTB entry. Return instructions are a special case where a RAS (Return Address Stack) provides the return address. Otherwise, the default next PC is current PC plus one. After generating (predicting) the next instruction address, the fetch engine accesses the memory hierarchy to fetch the instructions into the pipeline.

In the ID stage, the fetched instruction is decoded first. Registers are renamed. The source register is renamed by checking the renaming table. The output register gets a new physical register from the free list and the corresponding entry in the renaming table

is updated. Then, this renamed instruction is dispatched into the instruction issue queue to wait for issuing.

In the IS stage, ready instructions are selected to move to EX stage, if the corresponding functional units are available and issue logic has enough bandwidth.

In the RR stage, the issued instructions read values from the integer or floating point register files or the forwarding paths.

In the EX stage, the ALU or the Floating-Point Unit executes instructions. The execution latency is dependent on the type of instruction. Typically, division is the most expensive instruction. Load instructions have variable latencies that depend on whether they hit or miss in the L1 or the L2 cache.

In the WB stage, the outcome of EX stage is written back to the register file. Also, the mispredicted branches are recovered in this stage.

In the RE stage, the instruction retires if it is safe. If the exception bit is set for this instruction, recovery operations take place. All instructions after the exceptional instruction are flushed and refetched.

5.2.2 Potential performance bottlenecks

There are many factors that can lead to performance loss. Based on whether the performance constraints are hardware or software dependent, we classify those constraints into two categories - capacity constraints and inherent constraints, which are listed in Figure 5.2. The capacity constraints are caused by limited hardware resources, such as cache size, or read/write ports. The inherent constraints are both hardware and software dependent. For example, branch misprediction rate is impacted by branch predictor configuration, as

	Capacity Constraints (hardware)		Inherent Constraints (hardware and software)
IF	• Branch predictor size • BTB size • RAS size • I-cache size • I-TLB size	• Fetch width • Number of predictions/cycle • Prediction latency • I-Cache/memory latency • I-TLB miss latency • # cache read misses inflight	• Branch prediction accuracy • I-cache miss rate • I-TLB miss rate
ID	• Inst buffer size • Renaming resources # physical registers # shadow map	• Decoding width • Dispatch width	
IS	• Instruction Queue size • Load/Store Queue size • ROB size	• Issue width	• # inst issued/cycle
RR	• # read port of reg file		
EX	• # Int Unit • # FP Unit • # LD/ST Unit • D-cache size • D-TLB size	• Exe latency • Branch misprediction penalty • D-cache/memory latency • D-TLB miss latency • # cache read misses inflight	• D-cache miss rate • D-TLB miss rate
WB	• # write port of reg file		
RT		• Commit width • # cache write misses inflight	

Figure 5.2: Major performance constraints

well as the branch characteristics of the program.

5.3 Counter-based performance bottleneck modeling

In this section, we present an event counter based performance model. For modern microprocessors, event counters are becoming a standard on-chip resource. Utilizing event counters is a low overhead mechanism to monitor the behavior of microprocessor execution. However, for modern complex microarchitectures, event counters are not accurate enough for fine-grain instruction-level performance analysis because of the overlapping of events and the interaction between instructions. But, from a long-term program performance point of view, the overlapping effect can be approximately modeled with the event counter statistics.

The goal of our counter-based performance model is to provide runtime performance information to identify the performance bottlenecks that a microprocessor suffers. As we described in Section 5.2.2, there are many constraints that impact performance. With the consideration of hardware cost/effectiveness, investing the limited event counters to those critical events is a reasonable choice. For our generic superscalar microarchitecture, the critical events we choose are: I-TLB miss, IL1 miss, IL2 miss, direct unconditional branch mispredictions, issue queue being full, resource contention, expensive instructions, D-TLB miss, DL1 miss, DL2 miss and rest of branch mispredictions. While other events may be critical for a specific architecture or execution situation, these critical events we define are enough to reveal the general performance behavior. In addition, our analysis algorithm is generic. It can be extended and applied to new cases.

	Events	Cost Model
IF	I-TLB miss	$N * P_{itlb\text{-}miss}$
	IL1 miss	$N * P_{il1\text{-}miss}$
	IL2 miss	$N * P_{il2\text{-}miss}$
ID	Direct Branch misprediction	$N * (P_{misbp\text{-}dir})$
IS	Queue Full	N – Cost(exe)
EX	Resource Contention	N / f
	Execution Latency (Mul, Div…)	N * (latency - 1) / f
	D-TLB miss	$N * P_{dtlb\text{-}miss} / f$
	DL1 miss	$N * P_{dl1\text{-}miss} / f$
	DL2 miss	$N * P_{dl2\text{-}miss} / f$
WB	Branch Misprediction	$N * (P_{misbp\text{-}indir})$

Figure 5.3: Performance model

5.3.1 Long-term event cost model

The counters give the numbers of critical events for a sampling interval. However, these numbers can not tell us how important each of them is from the whole microprocessor performance viewpoint. For example, we should treat L1 cache misses and L2 cache misses differently, because the miss latency difference between them is almost 10 fold. It is clear that different critical events have a different impact on the overall microprocessor performance. Our model tries to translate those raw numbers of different events into a quantitative representation of performance. The modeling of each event is shown in Figure 5.3.

- I-TLB miss: An I-TLB miss causes a fetch engine stall. The cost of I-TLB miss events is the product of the number of events and the I-TLB miss penalty, because

when instruction fetch is stalled, a pipeline bubble will be created during the waiting period.

- I-L1 miss: The cost of I-L1 misses is the product of the number of events and the I-L1 miss penalty.

- I-L2 miss: The cost of I-L2 misses is the product of the number of events and the I-L2 miss penalty.

- Direct branch mispredictions: An unconditional direct branch misprediction causes a pipeline flush. The cost of direct branch mispredictions is the product of the number of events and branch misprediction penalty. The reason is the ID stage is still in in-order and a pipeline bubble will be created during the period of waiting for pipeline recovery.

- Resource Contention: Some instruction types will be stalled if there are not enough idle functional units. The cost of resource contention is the event number divided by a parallelism coefficient `f`, because resource contention doesn't cause a whole pipeline stall. Other type ready instructions can execute without waiting. So the performance impact of resource contention is proportional to the number of contention events and inversely proportional to the ILP. Since IPC is an approximation of available ILP, the coefficient `f` we choose is IPC plus one. The addition is to avoid amplifying the event cost when IPC is less than 1.

- Expensive Instructions: Some expensive instructions, such as division, take a longer time to finish. The cost of expensive instructions is the product of the number of

events and the extra execution latency divided by the parallelism coefficient `f`.

- D-TLB miss event: The cost of D-TLB misses is the product of the number of events and D-TLB miss penalty divided by parallelism coefficient `f`.

- D-L1 miss event: The cost of D-L1 misses is the product of the number of events and D-L1 miss penalty divided by parallelism coefficient `f`.

- D-L2 miss event: The cost of D-L2 misses is the product of the number of events and D-L2 miss penalty divided by parallelism coefficient `f`.

- Rest of branch mispredictions: Mispredicted conditional branches and indirect branches cause a longer instruction fetch engine stall. The cost is the product of the number of events and the misprediction penalty.

- Issue queue being full: The issue queue being full causes a pipeline stall. But the stall in EX stage is the major reason of issue queue being full. So the cost of issue queue being full is the number of events minus the cost of events in EX stage.

5.3.2 Model verification

The goal of our cost model is to quantify the performance effects of events based on the bubbles that are inserted into the pipeline. The straightforward way to verify our cost model is to compare the actual execution cycles to the sum of the execution cycles in an ideal pipeline (i.e. pipeline throughput without bubbles) and the event costs from our model. The formulas used for model verification are as follows.

$$T_{model} = T_{ideal} + \sum_{i} Cost_i \tag{5.1}$$

$$R_{error} = |T_{real} - T_{model}|/T_{real} \tag{5.2}$$

In Formula 5.1, the predicted execution time (`Tmodel`) from our model is computed by adding the ideal execution time (`Tideal`) and the total event cost (`Costi`). The ideal execution time is the number of executed instructions divided by the issue width, assuming each pipeline stage takes one cycle. Formula 5.2 represents the relative model error rate (`Rerror`) calculated by dividing the absolute value of the difference between real execution time and predicted execution time with the real execution time. Figure 5.4 shows the error rate for different execution periods - 1M, 10M, 100M and 1B instructions. On average, our model can achieve a 5% error rate, when l0M or more instructions are executed.

5.4 Performance bottleneck phase tracking

In the previous section, we presented an approach to collect system performance information. Now we discuss how to analyze performance bottlenecks based on that information.

5.4.1 Bottleneck vector

Choosing an appropriate quantitative representation is very important for data analysis. A good representation can make analysis much easier and more accurate. For our model, we choose a vector to represent the system performance information. There are two

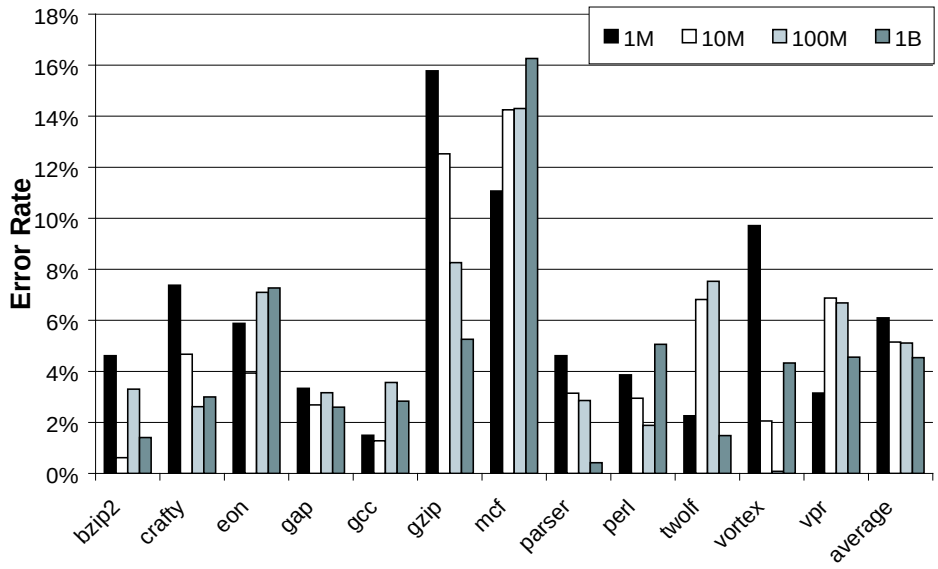

Figure 5.4: Performance model verification

reasons why we use a vector. The first is that the data we collect are disjoint values in event counters. These counters convey different information about different components in the microprocessor and will be handled separately. Therefore, keeping data in a vector will be a proper way without information loss. The other reason is that vectors are a powerful and fundamental mathematical representation. Computations based on vectors are relatively easy in both computation and hardware cost.

Based on the event counters we described in the last section, the bottleneck vector has eleven dimensions which corresponds to each counter. The vector design is shown in Figure 5.5.

5.4.2 Bottleneck phases

Researchers proposed the program phase concept to describe the phenomena that a program exhibits repeating long-term behavior. The principle behind program phases is executing instructions visiting the static program with regular patterns. For example, in a loop, the static loop body will be visited repeatedly during execution. It is not a surprise that program phase phenomena has a similar effect on performance bottlenecks because performance bottlenecks are determined by both hardware and software. With a fixed hardware configuration, regular software patterns result in regular bottleneck behavior, called bottleneck phases.

To illustrate the bottleneck phase behavior, we track the bottleneck vectors for 12 SPEC 2000 benchmarks for a 10B instruction execution, which is sampled every 1M instructions. Each element of the vector is shown in parallel in the behavior graph. The bottleneck intensity is represented by darkness. The darker the points are, the more intense the performance bottleneck is. Figure 5.6, 5.7, 5.8, 5.9 and 5.10 show each benchmark performance bottleneck behavior respectively.

- bzip2

  Bzip2 varies its behavior during execution. It suffers from issue queue being full, branch mispredictions and data cache misses intermittently.

- gap

  Gap exhibits very regular behavior. There are very few outstanding performance bottleneck during execution. Only in a very short period, it suffers from issue queue being full and data cache misses.

$$\vec{\mathbf{B}} = \begin{pmatrix} b_{rsc} \\ b_{queue} \\ b_{long-inst} \\ b_{rest-bp} \\ b_{dir-bp} \\ b_{dl2} \\ b_{dl1} \\ b_{dtlb} \\ b_{il2} \\ b_{il1} \\ b_{itlb} \end{pmatrix}$$

Figure 5.5: Bottleneck Vector

- gcc

  Gcc varies its behavior over the whole execution time. It suffers from issue queue being full, instruction cache misses, data instruction cache misses and branch misprediction for different periods.

- mcf

  Mcf suffers extremely from a high data cache miss bottleneck. But in the beginning, it is bottleneck-free.

- vpr

  Vpr suffers from data cache misses and branch misprediction bottlenecks at same time.

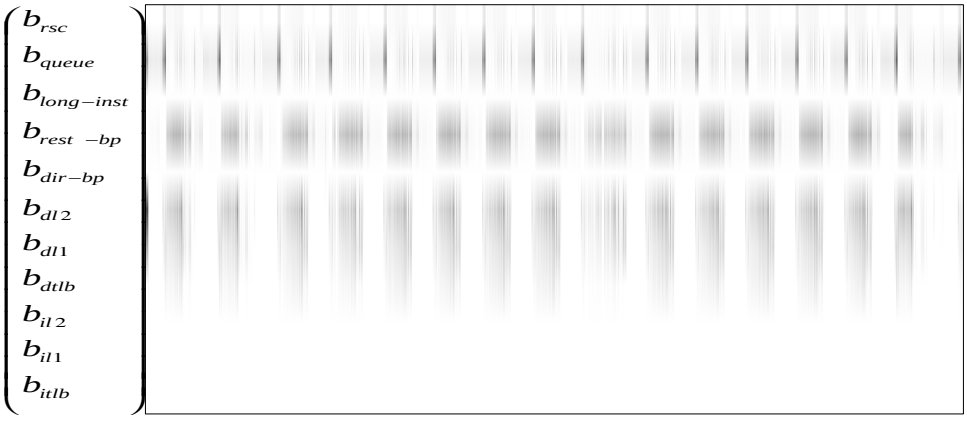

Figure 5.6: bzip2 performance bottleneck behavior

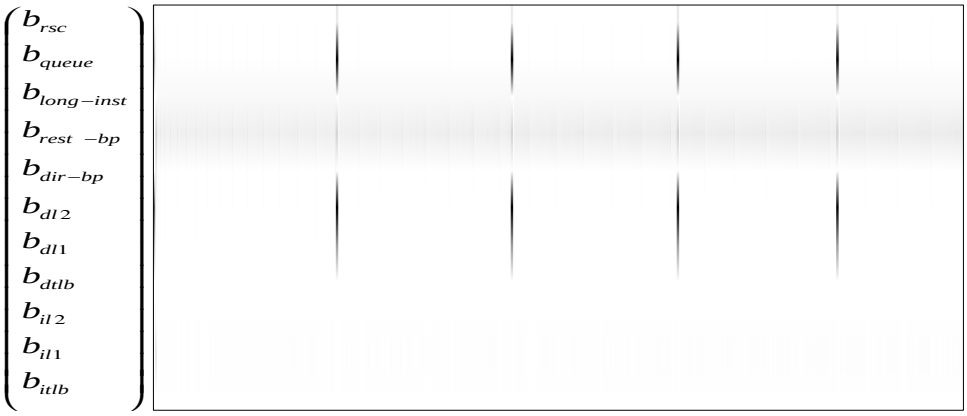

Figure 5.7: gap performance bottleneck behavior

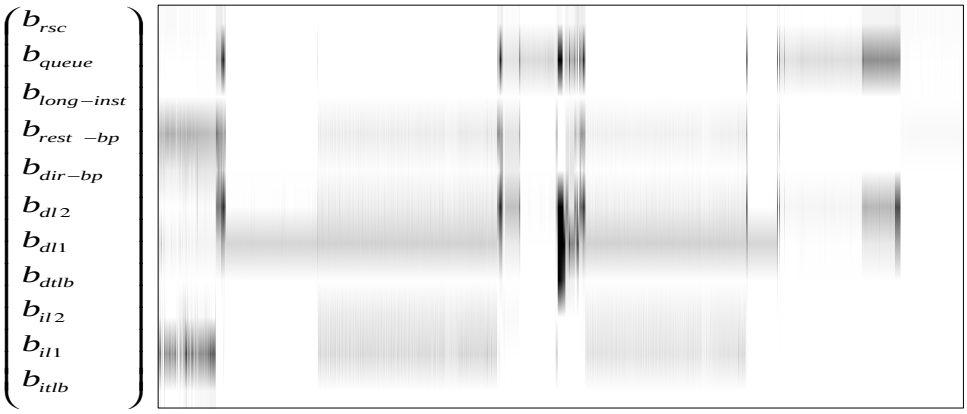

Figure 5.8: gcc performance bottleneck behavior

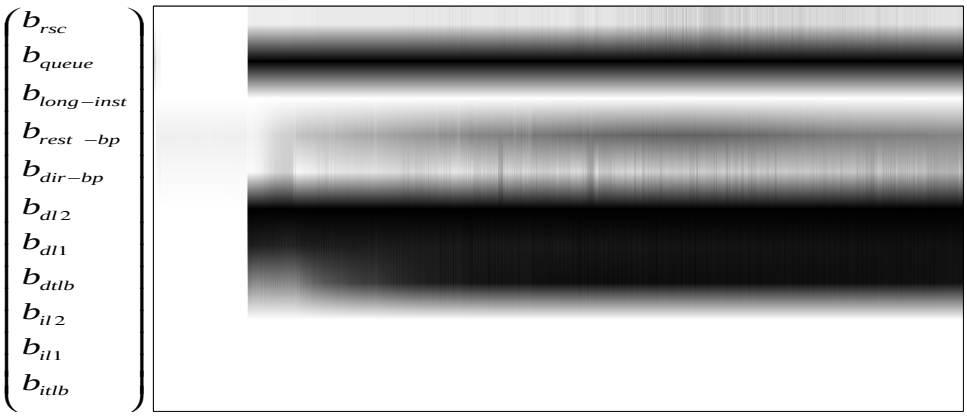

Figure 5.9: mcf performance bottleneck behavior

Figure 5.10: vpr performance bottleneck behavior

5.4.3 Bottleneck phase tracking

Identifying bottlenecks is not a difficult task from a mathematical perspective. We can use the vector itself as an ID. However, hardware cost and computation complexity will make this design very expensive. One efficient way to keep component cost information without involving too much hardware budget is hashing the elements of the bottleneck vector into a phase ID. The tracking scheme is shown in Figure 5.11. For a given sampling interval, event counters are sampled and cleared. Then those raw event numbers are processed by an array of function blocks. The function blocks perform two computations. One is average cost computation based on the cost model. In our study, we use cost per 1K instructions as the average cost for a sampling interval. The other is cost normalization computation, which is translating the cost from the first step into the number of cost units. The cost unit is a fixed cost window defined by users, depending on the accuracy requirement. The reason for using the cost unit is to classify similar cost values into the same cost group. Finally, the elements of the processed bottleneck vector are hashed into a bottleneck phase ID.

5.5 Performance bottleneck phase prediction

In the previous section, we described how to track and identify a bottleneck phase at runtime. There may be a need to predict bottlenecks in advance to guide system optimization. In this section we will present the predictability of performance bottlenecks with several prediction schemes.

As we described in Section 5.4.3, bottleneck phase IDs depend on both the cost

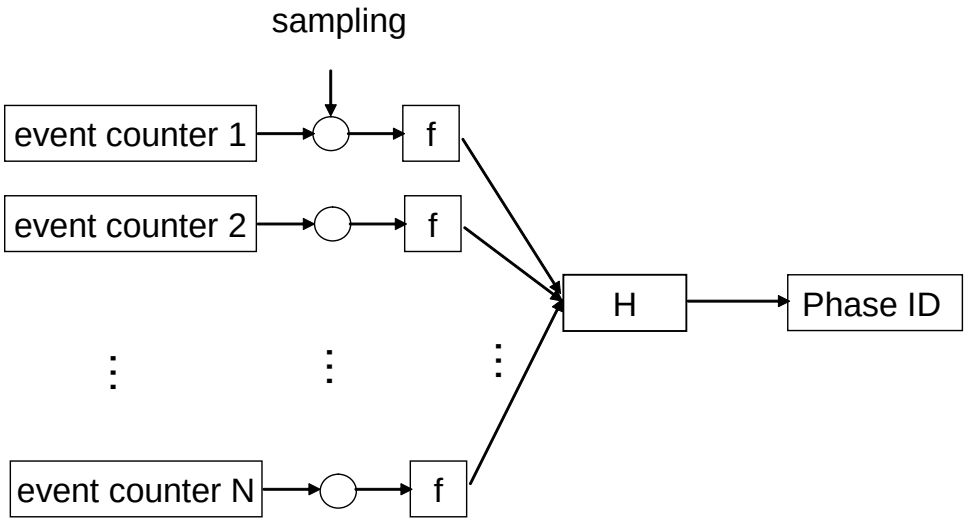

Figure 5.11: Bottleneck phase tracking architecture

unit and the hash function. The smaller the cost unit is, the more phase IDs there are. With an increasing number of phases, prediction becomes more difficult. A last value predictor is the simplest prediction scheme, which predicts the current phase as the next phase. The last value predictor relies on phase stability. In Figure 5.12, we plot its prediction accuracy and the number of phases for several cost units. We can see that as the cost unit increases, the number of phases decreases and the corresponding prediction accuracy increases steadily.

In addition to a last value predictor, we studied two other prediction schemes - history predictor and Markov predictor. The history predictor keeps the previous phase information and predicts the phase with the most number of appearances in the history as the next phase prediction. The history predictor can filter sudden phase change noises, providing more stable predictions than the last value predictor if there are frequent phase transitions. The third prediction scheme we studied is a Markov predictor. We use history information to identify the transition state. The Markov table is indexed by the lower bits

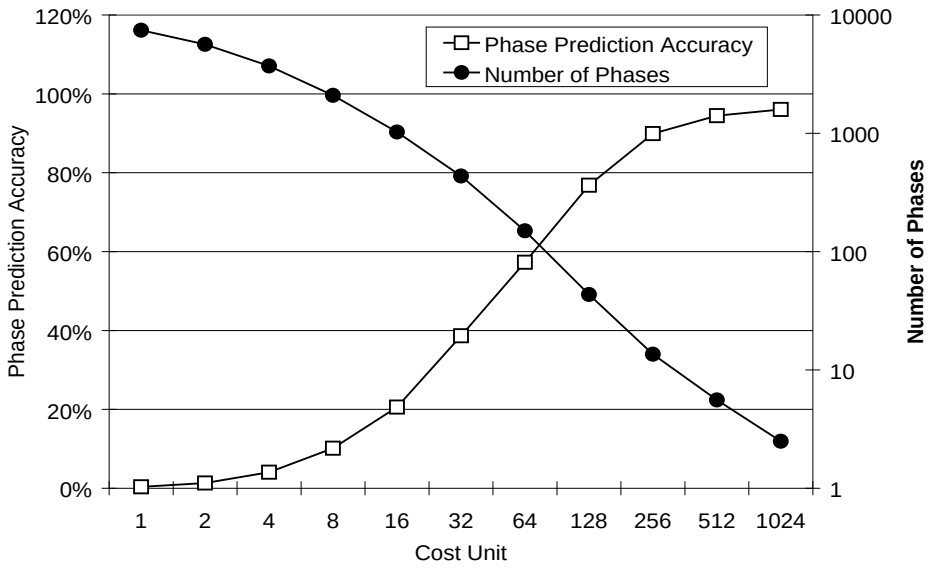

Figure 5.12: Prediction accuracy vs. phase ID numbers

of the hashed history information. Each entry consists of the higher bits of the hashed history information as the tag and the predicted phase ID. For every prediction, Markov prediction table is checked first. On a hit, the phase ID stored in this entry will be the prediction. On a miss, we use the current ID as the prediction. In Figure 5.13, we compare these predictors. The cost unit is 64 for this figure. The sampling period is 1M instructions and 10B instructions are executed in total. The history length for the history and Markov predictors is 3. The Markov table has 256 entries. We can see that for most of the benchmarks, the three predictors get similar accuracies. The reason is that the bottleneck phases are fairly stable. For `mcf`, since the numbers of phases and transitions between phases are high, the Last predictor gives extremely low prediction accuracy and the Markov predictor provides much more accurate predictions.

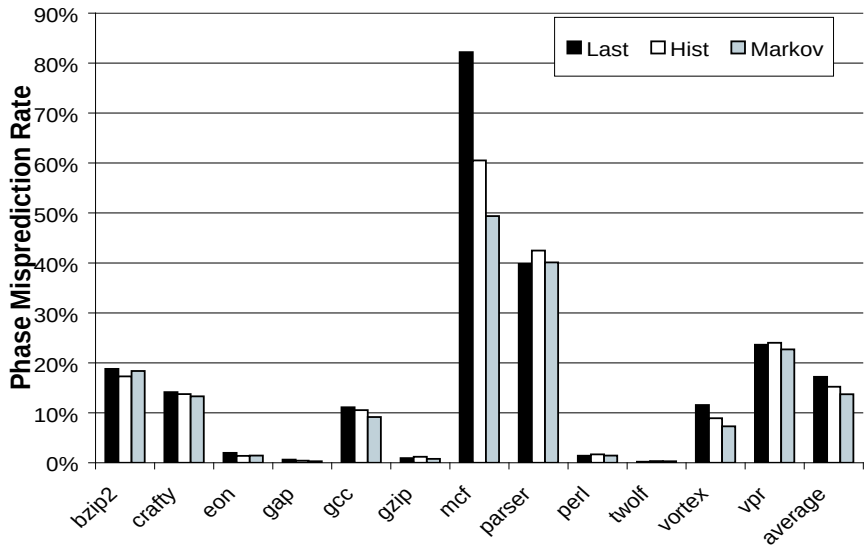

Figure 5.13: Bottleneck phase predictors

Chapter 6

Adaptive Value Predictor

This chapter presents an application of the CiC model - Adaptive Value Predictor (AVP). AVP extends the original value predictor functionality to attack various performance bottlenecks at runtime. The following sections discuss AVP in detail.

6.1 Motivation

Modern general-purpose microprocessors are getting complex and equip more "help engines" to exploit more ILP/TLP. The increasing complexity makes microprocessor's performance hard to understand. Meanwhile, research shows that different applications have different characteristics. For instance, some of them are computation intensive, some of them are memory intensive. Actually, the runtime performance bottleneck depends on both of hardware and software. To understand performance bottlenecks in superscalar microarchitectures better, we run simulations when a specific processor component is idealized. The applications are from SPEC 2000 INT benchmarks. The microarchitecture is a 4-way

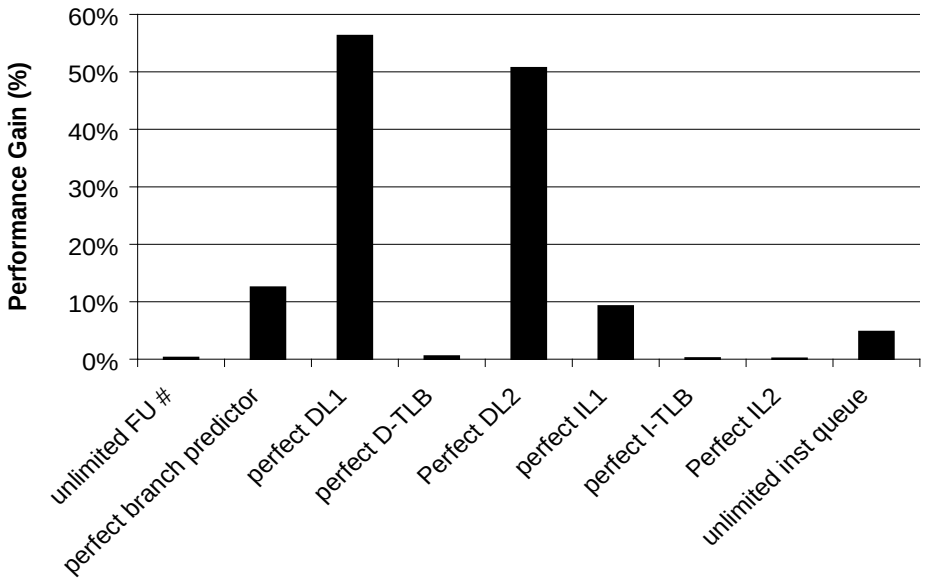

Figure 6.1: Performance gain with ideal components

superscalar with specific parameters described in Chapter 4. Figure 6.1 shows results for:

- Unlimited number of function units

  With an unlimited number of function units, the performance gain is not noticeable. That means for most applications the function units is not a major limit of performance.

- Perfect branch prediction

  With perfect branch prediction, we can get a 12% performance gain. That means the branch prediction is one of the major performance bottlenecks.

- Ideal L1 data cache

  It is not a surprise that a perfect DL1 cache achieves more than 50% performance gain. That means the original applications suffer from long memory access latencies

significantly.

- Ideal D-TLB

  Ideal D-TLB does not result in much performance benefit. That means for most applications commonly used data reside in a few pages and the D-TLB can provide the address translation most of time.

- Ideal L2 data cache

  Since L2 access is caused by part of L1 accesses (i.e. L1 misses), the ideal L1 data cache performance covers ideal L2 data cache performance. We can see the ideal L2 data cache performance gain is a major part of the ideal L1 data cache performance gain. Therefore, we conclude that L2 cache access is the dominant component of the of performance bottleneck caused by memory accesses for fetching data.

- Ideal L1 instruction cache

  Ideal IL1 cache can give about 10% performance benefit. It means instruction cache is one of potential performance bottlenecks.

- Ideal I-TLB

  Ideal I-TLB does not cause noticeable performance improvement. It means instruction address translation is not a performance bottleneck.

- Ideal L2 instruction cache

  Ideal L2 instruction cache does not provide noticeable performance gain. Then we can conclude that the L1 instruction cache contributes significantly to bottleneck caused by instruction memory accesses.

- Size of issue queue

  Unlimited issue queue gives a 5% performance gain. It means the issue queue is one of potential performance bottlenecks.

From Figure 6.1, we can see that the branch prediction, IL1 cache, DL2 cache and issue queue size are general performance limits. There are several approaches to attack those performance bottlenecks individually. Traditional value prediction is an effective technique to eliminate data flow restrictions by predicting register values before they are resolved [36]. Programs exhibit value locality, i.e. recurrence of register values. Value prediction exploits value locality to predict register values with a predictor, in order to break data dependences among instructions and provide more available instructions to improve ILP. Meanwhile, Gonzalez et al. [22] proposed control-flow speculation through value prediction for superscalar processors. In their work, they predict the outcomes of branches by predicting the value of inputs with a value predictor and performing an early computation of results according to the predicted values. In addition, researchers also use value prediction to assist prefetching, in order to tolerate long memory access latencies [27, 11].

Since fairly large value prediction tables are needed for high accuracy, it is infeasible to include separate value predictors in hardware to take advantage of all these techniques. There is an alternative adaptive design however: an adaptive value predictor, which can work for branch prediction, value prediction or prefetching. The key of an adaptive value predictor is how to to tune its functionality to program needs. We propose a runtime performance bottleneck analysis model as the basis of the adaptation algorithm. When

the performance bottleneck is identified as the control dependences in the program and the branch predictor can not provide accurate prediction, the value predictor can help the branch predictor to improve branch prediction accuracy. When the instruction memory accesses are the performance bottleneck, the value predictor can predict instruction memory addresses to prefetch. When the data memory accesses are the performance bottleneck, the value predictor can predict data memory addresses to prefetch. When the issue queue becomes a performance bottleneck, the value predictor works in its original functionality - traditional value prediction to break up data dependence chains.

6.2 Traditional value predictor

True data dependencies can greatly impede instruction level parallelism (ILP). Data dependencies decrease ILP when long latency instructions flow through the pipeline, and there are not enough independent instructions available to keep the processor busy. Value prediction is an approach that breaks true data dependency chains by predicting the resulting value for an instruction, and by allowing dependent instructions to use this predicted value as a source value. This allows the dependent instructions to execute in parallel.

Figure 6.2 shows the performance gain of a two-delta stride value predictor. It improves performance by 20% on average.

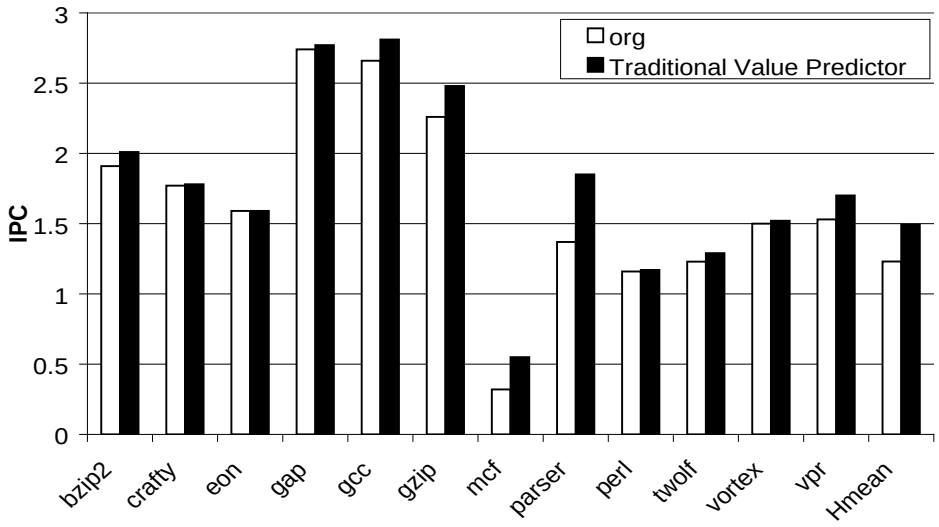

Figure 6.2: Performance of traditional value predictor

6.3 Branch predictor assistant

Branch prediction is very important for a modern microprocessor's performance, as pipelines are getting deeper. Gonzalez [22] proposed a branch prediction scheme - Branch Predictor through Value Prediction (BPVP). This branch predictor predicts the outcomes of branches by predicting the value of their inputs and performing an early computation of their results according to the predicted values. A chooser chooses the final prediction from a general prediction and the precomputed prediction.

Figure 6.3 shows the branch prediction accuracy of a *gshare* branch predictor without and with value prediction. The BPVP improves conditional branch prediction accuracy by about 2% on average.

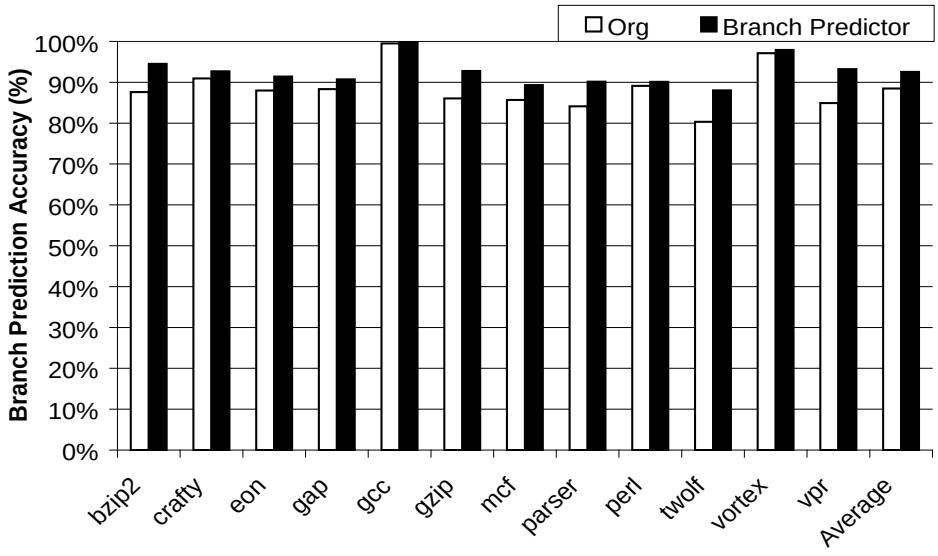

Figure 6.3: Branch prediction accuracy improvement with BPVP

6.4 Data prefetching

The speed gap between CPU and memory continues increasing. Data prefetching is an effective technique to tolerate memory latency. Normally, the processor fetches data from memory when there is a load instruction, called demand-based data fetching. If the data is missed in the cache, a request is made to the external memory system. By comparison, data prefetching attempts to fetch data before the processor requests that data. It predicts the future memory requests of the processor and issues those requests early to improve data fetch latency.

Figure 6.4 shows the L2 cache miss rate impact of a two-delta stride L2 prefetcher. We can see L2 prefetcher reduces 50% L2 misses on average.

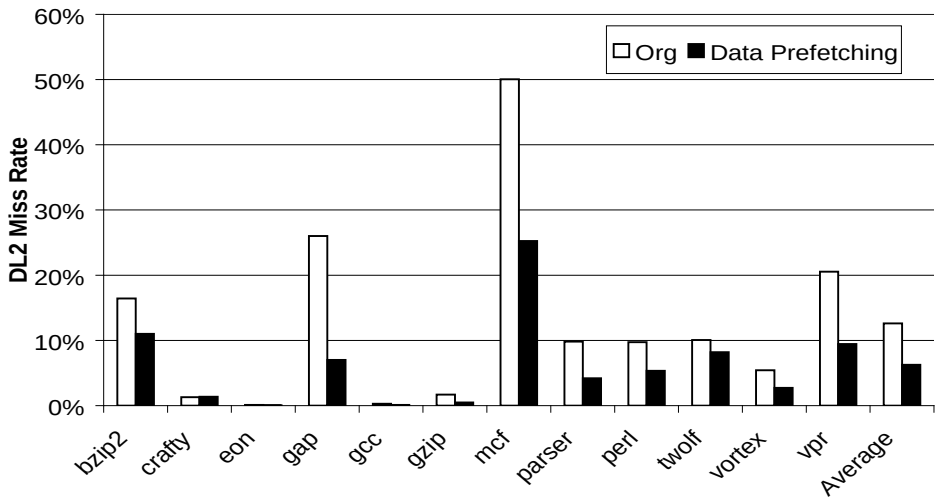

Figure 6.4: DL2 cache miss rate

6.5 Instruction prefetching

A modern super-scalar microprocessor can be divided into three engines - in-order fetching engine, out-of-order execution engine and in-order retire engine. The in-order fetching engine consists of IF (Instruction Fetch) and ID (Instruction Dispatch) stages typically. It prepares instructions for following execution engine. Since it is in the first stage of pipeline, the slowdown in fetching engine will impact the performance directly. Instruction cache is a major potential performance bottleneck besides branch prediction in fetching engine. Instruction cache misses are usually more expensive than data cache misses as instruction misses stall the processor pipeline, while a significant proportion of data misses can often be overlapped with other instructions in an out-of-order execution engine. Actually, for SPEC 2000 benchmark set we study in this work, instruction cache misses is not so intensive. The instruction miss rate is about 0.7% on average as shown in Figure

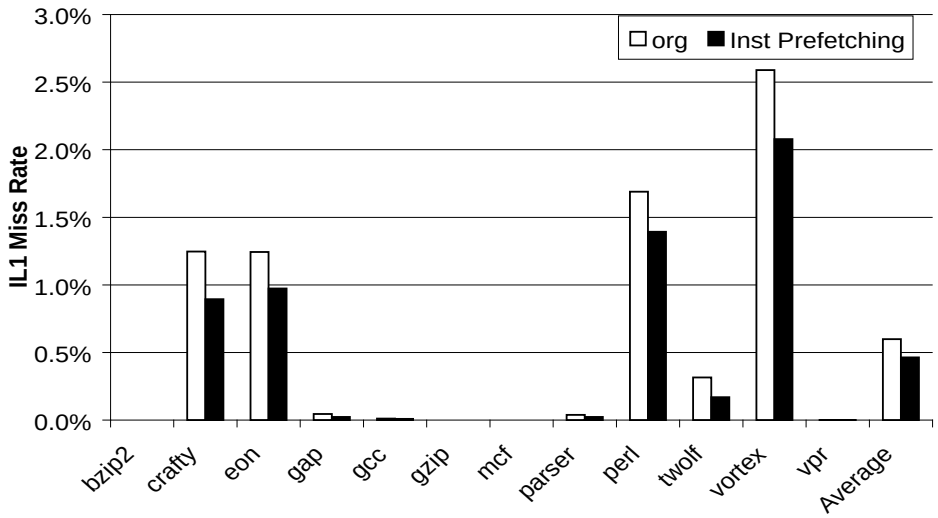

Figure 6.5: IL1 cache miss rate

6.5. Many commercial workloads (a database workload, TPC-W, SPECjAppServer2002 and SPECweb99) have more than 10% instruction cache miss rate.

In our work, we use two-delta stride prefetcher to catch both sequential and non-sequential instruction streams. Figure 6.5 shows the cache miss rate improvement. The instruction prefetcher reduces 23% of instruction cache misses on average.

Chapter 7

AVP CiC Design

For this particular CiC feedback control model, the adaptive unit will be the value predictor, which has four functionalities - traditional value predictor, branch predictor, data prefetcher and instruction prefetching. As we discussed in Chapter 3, there are two basic components comprising the CiC control unit - the monitor and the controller. For the AVP, the monitor is in charge of collecting system performance information. The controller is in charge of choosing functionality of the value predictor. In the following sections, we will discuss the monitor and the controller designs in detail.

7.1 AVP monitor

The idea of AVP is to tune the functionality of the value predictor to the performance bottlenecks. The monitor has to collect the performance bottleneck related information to help the controller make its decision. In Chapter 5, we presented a performance model and the mathematical representation of performance bottlenecks - the `bottleneck`

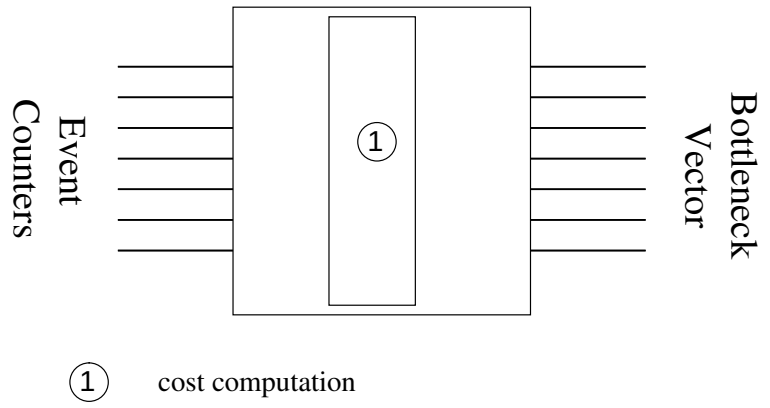

Figure 7.1: AVP Monitor

`vector`. Based on this model, the monitor can be implemented as shown in Figure 7.1. The input of the monitor is the raw system information from event counters, which reside in specific hardware components to count the corresponding events. The output of the monitor is the bottleneck vector. The monitor implements the computation logic to translate the raw system information into bottleneck vector based our performance model.

7.2 AVP controller

The controller decides the functionality of the adaptive value predictor based on the bottleneck vector - the output of the monitor. The output of the controller is one of 4 adaptation choices: traditional value predictor, branch predictor, instruction prefetcher or data prefetcher.

7.2.1 Controller operations

There are 4 steps involved in the controller operations as shown in Figure 7.2.

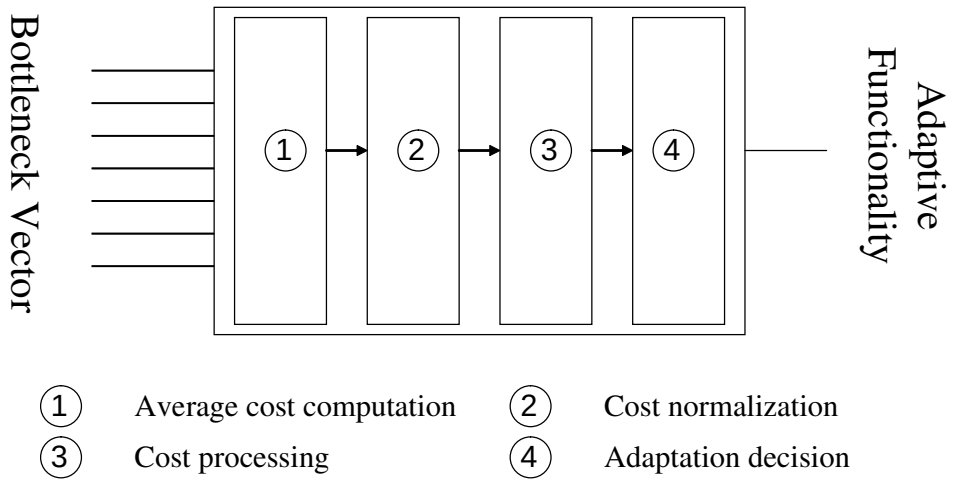

Figure 7.2: AVP Controller

1. **Average cost computation**

   The input frequency of the controller depends on the sampling period of the monitor. The choice of sampling period is application dependent. For performance bottleneck analysis, using average cost is an appropriate way to reveal the program performance behavior.

2. **Cost normalization**

   In order to improve the stability of the AVP and simplify the cost processing, we group close cost values into quantized cost levels by defining a cost unit. After this step, the averaged cost values are reported in terms of the cost unit. The value of the cost unit depends on the cost accuracy requirements.

3. **Cost processing**

   For the adaptation of the AVP, there are 4 categories of costs corresponding to its 4 functions. These are instruction cache access costs, data cache access costs, the cost

of issue queue being full and branch misprediction costs. Instruction cache access cost is the sum of ITLB miss cost, IL1 cache miss cost and IL2 cache miss cost. Data cache access cost is the sum of DTLB miss cost, DL1 miss cost and DL2 cache miss cost. Branch misprediction cost is the mispredicted branch costs in ID and WB stages.

4. **Adaptation decision**

After getting the related cost information, the controller will choose the most beneficial functionality for the AVP. We propose three adaption algorithms for this purpose: greedy, history, and conservative algorithms. The greedy algorithm is shown in Algorithm 1. It is a straightforward adaptation algorithm that changes the functionality of the AVP immediately after detecting a new performance bottleneck. The advantage of the greedy algorithm is fast response. The disadvantage is the stability of the AVP. Algorithm 2 shows the history algorithm. It transfers functionality only when it sees repeating performance bottlenecks. The history algorithm can filter noise transitions to increase the AVP stability. However it is not as fast as the greedy algorithm to response to a performance bottleneck change. The conservative algorithm is another adaptation algorithm to increase the stability of AVP. It is shown in Algorithm 3. It considers the performance loss when transferring to another functionality. It changes the state only when it is sure that the benefit of a new functionality is greater than the performance loss because of giving up the current functionality.

Table 7.1: Notes for algorithm description

Word	Description
AVP	AVP state
VP	AVP state - traditional value predictor
BP	AVP state - branch predictor
IPF	AVP state - instruction prefetcher
DPF	AVP state - data prefetcher
InstFetchCost	instruction cache cost
DataFetchCost	data cache cost
MisBpCost	mispredicted branch cost
IqueueCost	instruction queue being full cost
MaxCost	maximum cost among 4 cost categories
Max(...)	a function to find maximum cost
Decision	AVP functionality decision
PredDecision	predicted AVP decision
LastPredDecision	last predicted AVP decision
RepeatingNum	repeating number of current state
RepeatingThreshold	repeating number threshold
TmpCost	temporary variable
ThresholdCost	cost threshold for conservative decision

7.2.2 Adaptation algorithm

This section describes the three AVP adaptation algorithms. For AVP's adaptation, there is a tradeoff between fast response and stability. Fast response can tune AVP's state to performance bottleneck changes in a timely fashion. However, too frequent state transitions do not help because state transition is not free as the prediction table needs training in order to give accurate predictions. The algorithms we propose in this section are at different points on this tradeoff curve. We will discuss them in detail in following subsections. To represent these algorithms clearly, we also present them in pseudo code. The variable names used in these pseudo codes are explained in Table 7.1.

Algorithm 1 - Greedy

The idea of the greedy algorithm is immediately transferring the AVP's state after detecting a performance bottleneck change. It is closer to the fast response side in the tradeoff curve between fast response and stability. When there are few steady performance bottleneck changes, the greedy algorithm can change AVP's state in a timely manner. However, if there are many unstable performance bottlenecks and the greedy algorithm changes its state too frequently, AVP will not help and can even hurt performance due to inaccurate predictions and the related recovery costs.

The algorithm is written in pseudo code in Algorithm 1. After every sampling interval, we compare the cost value for these 4 performance bottlenecks and find the maximum cost value. If the instruction cache access cost is the maximum, next AVP state will be an instruction prefetcher. If data cache access cost is the maximum, next AVP state will be an data prefetcher. If branch misprediction cost is the maximum, next AVP state will be a branch predictor. If the cost of issue queue being full, next AVP state will be a traditional value predictor.

Algorithm 1 Greedy

```
1: MaxCost ← Max(InstFetchCost, DataFetchCost, MisBpCost, IqueueCost)
2: if IqueueCost == MaxCost then
3:     Decision ← VP
4: end if
5: if DataFetchCost == MaxCost then
6:     Decision ← DPF
7: end if
8: if MisBpCost == MaxCost then
9:     Decision ← BP
10: end if
11: if InstFetchCost == MaxCost then
12:     Decision ← IPF
13: end if
14: AVP ← Decision
```

Algorithm 2 - History

The idea of the history algorithm is changing the AVP state only after detecting a steady performance bottleneck change. In contrast to the greedy algorithm, it is closer to the stability in the tradeoff curve between fast response and stability. The main target of the history algorithm is eliminating "noise performance bottleneck transitions". The scenario of a "noise transition" is when there is a steady performance bottleneck, with only few other bottlenecks embedded in the steady performance bottleneck in the background. These few different bottlenecks are called "noise performance bottlenecks" for a steady performance bottleneck. For those "noise performance bottlenecks", the good choice is ignoring them to keep the AVP state stable for prediction accuracy.

The history algorithm is written in pseudo code in Algorithm 2. In this algorithm the controller has to remember the last prediction state (i.e. the performance bottleneck in the last sampling period) and the repeating frequency of this state. Like greedy, it finds the maximum cost component and corresponding bottleneck category. Then the adaptation decision is made. Finally, we update the repeating number of the predicted decision. If this decision is the same as the last decision, we increment the counter. If this value of repeating counter is greater than a preset repeating threshold and current decision is different from the current AVP state, we change the current AVP state to a new corresponding one. If current prediction is different from the last prediction, it resets the repeating counter and waits for the next processing cycle.

Algorithm 2 History

```
1: MaxCost ← Max(InstFetchCost, DataFetchCost, MisBpCost, IqueueCost)
2: if IqueueCost == MaxCost then
3:     if RepeatingNum ≥ RepeatingThreshold and LastPredDecision == VP then
4:         Decision ← VP
5:     end if
6:     PredDecision ← VP
7: end if
8: if DataFetchCost == MaxCost then
9:     if RepeatingNum ≥ RepeatingThreshold and LastPredDecision == DPF then
10:        Decision ← DPF
11:    end if
12:    PredDecision ← DPF
13: end if
14: if MisBpCost == MaxCost then
15:    if RepeatingNum ≥ RepeatingThreshold and LastPredDecision == BP then
16:        Decision ← BP
17:    end if
18:    PredDecision ← BP
19: end if
20: if InstFetchCost == MaxCost then
21:    if RepeatingNum ≥ RepeatingThreshold and LastPredDecision == IPF then
22:        Decision ← IPF
23:    end if
24:    PredDecision ← IPF
25: end if
26: if PredDecision == LastPredDecision then
27:    RepeatingNum ← RepeatingNum + 1
28: else
29:    RepeatingNum ← 0
30: end if
31: LastPredDecision ← PredDecision
32: AVP ← Decision
```

Algorithm 3 - conservative

The idea of the conservative algorithm is changing the AVP state with the consideration of the potential performance loss because of AVP giving up the current functionality. It is another algorithm towards stability in the tradeoff curve between fast response and stability. The main target of smart decision is "state oscillation". "State oscillation" occurs when after the AVP changes to a new state, the previous performance bottleneck hurts performance again in the next interval. Then the AVP changes state back to the old state, and this keeps repeating. The AVP will then oscillate between these states. That will hurt performance because the prediction table does not have enough time to train and give accurate predictions. For "state oscillation", staying in the same state may be a better choice.

The algorithm of conservative algorithm is written in pseudo code in Algorithm 3. Like the greedy algorithm, it finds the maximum cost component and corresponding bottleneck category. It remembers the cost value of the bottleneck category that AVP target at currently. If the new bottleneck category is different from the current bottleneck category, it compares the cost value of the new bottleneck category and the cost value of the current bottleneck category. If the difference is greater than a preset threshold value, the AVP will change state at next interval. Otherwise, it will stay in the current one.

7.3 AVP default configurations

In this section, we present the default AVP configurations. If there is difference for an experiment, only the different part will be mentioned. The others will keep their

Algorithm 3 Conservative

```
1: if AVP == VP then
2:    TmpCost ← IqueueCost
3: else if AVP == BP then
4:    TmpCost ← MisBpCost
5: else if AVP == IPF then
6:    TmpCost ← InstFetchCost
7: else
8:    TmpCost ← DataFetchCost
9: end if
10: MaxCost ← Max(InstFetchCost, DataFetchCost, MisBpCost, IqueueCost)
11: if IqueueCost == MaxCost then
12:    if (IqueueCost − TmpCost) ≥ ThresholdCost then
13:       Decision ← VP
14:    end if
15: end if
16: if DataFetchCost == MaxCost then
17:    if (DataFetchCost − TmpCost) ≥ ThresholdCost then
18:       Decision ← DPF
19:    end if
20: end if
21: if MisBpCost == MaxCost then
22:    if (MisBpCost − TmpCost) ≥ ThresholdCost then
23:       Decision ← BP
24:    end if
25: end if
26: if InstFetchCost == MaxCost then
27:    if (InstFetchCost − TmpCost) ≥ ThresholdCost then
28:       Decision ← IPF
29:    end if
30: end if
31: AVP ← Decision
```

Table 7.2: Adaptive value predictor configurations

value predictor	16K entries PC indexed two-delta stride value predictor
CiC monitor	sampling at 100K committed instruction period
CiC Controller	cost per 1K committed instructions cost unit is 10 default state repeating threshold for history algorithm is 2 default state transition threshold for conservative algorithm is 5

default values. The default AVP configurations are shown in Table 7.2.

Chapter 8

Results

Now that we have described the AVP design, in this chapter, we will evaluate the different design options quantitatively.

8.1 Adaptive vs. non-adaptive

The AVP adds the flexibility to be able to tune its functionality to the runtime performance bottleneck when compared to the traditional value predictor. In this section, we present the effectiveness of the AVP by comparing its performance to non-adaptive value predictor designs.

8.1.1 Speedup

Figure 8.1 shows the IPC results of the AVP with the greedy algorithm. We compare them to the non-adaptive designs: baseline design (without value predictor), traditional value predictor, branch predictor, instruction prefetcher and data prefetcher. Figure

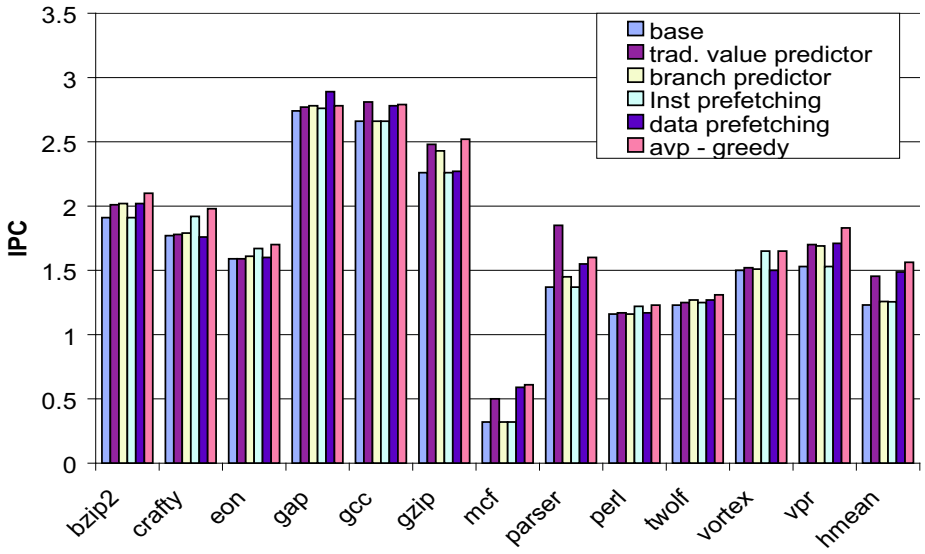

Figure 8.1: Performance results of the AVP with the greedy algorithm

8.2 shows the speedup results that is normalized to the result of baseline design. We can see that the AVP gives the best performance or close to the best performance for each benchmark. It gets 10% performance gain compared to the traditional value predictor and 30% compared to the baseline design.

Figure 8.3 shows the execution breakdown of 4 functionality choices. We can clearly see the adaptive value predictor tuning its functionality to the characteristics of each benchmark.

The detailed performance of AVP for each benchmark is described as follows.

- **bzip2**

  Bzip2 varies performance bottlenecks during program execution in three bottleneck types - issue queue being full, mispredicted branch and data cache misses. The AVP changes its functionality between traditional value predictor, branch predictor and

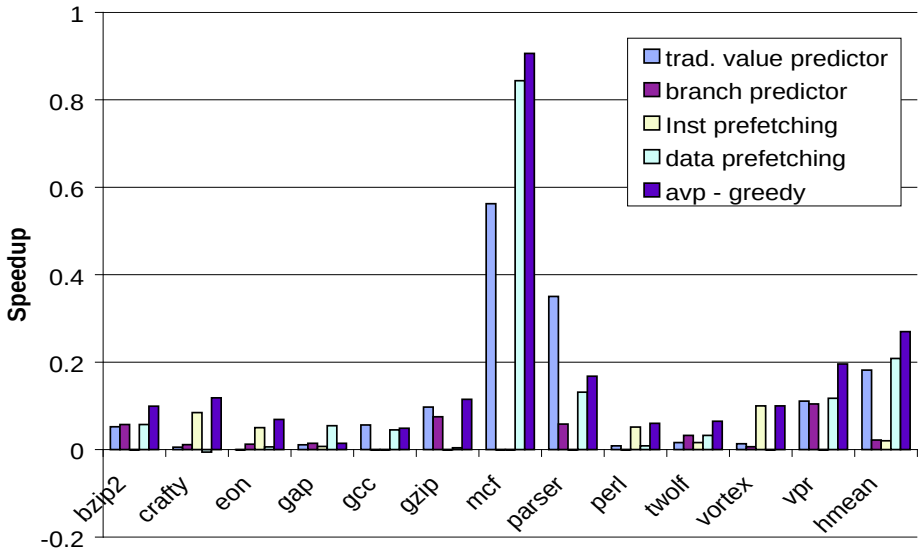

Figure 8.2: Speedup of the AVP with the greedy algorithm

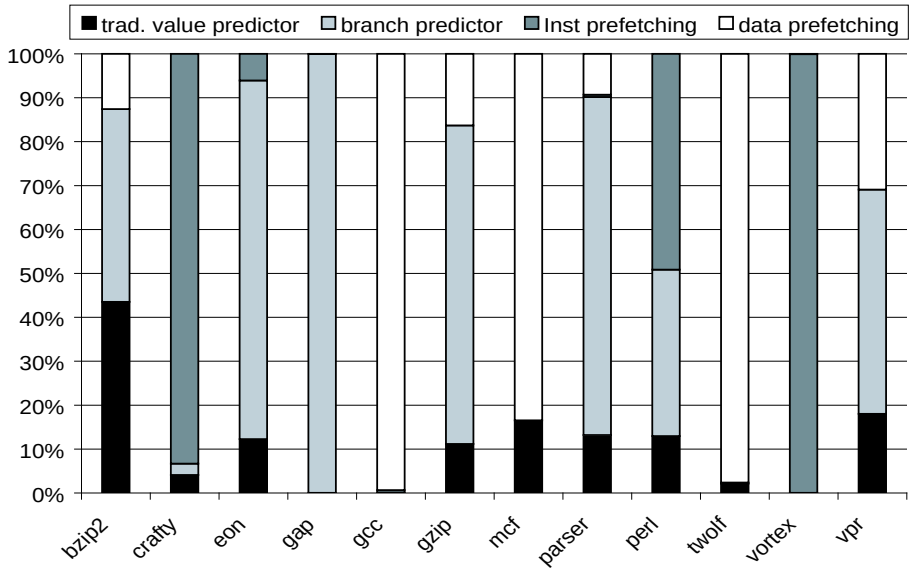

Figure 8.3: Execution breakdown of 4 function choices

data prefetcher. We can see the AVP outperforms any non-adaptive design.

- **crafty**

  Crafty suffers instruction cache misses dominantly during program execution. The instruction prefetcher outperforms other non-adaptive design. The AVP keeps the benefit of instruction prefetcher. In addition, it gets extra benefit from the value predictor and branch predictor assistant functionalities for other performance bottleneck phases.

- **eon**

  Eon has three performance bottlenecks during program execution - issue queue being full, mispredicted branch and L1 instruction cache misses. The AVP changes its functionality between value prediction, branch predictor assistant and instruction prefetcher. The AVP gets a significant performance gain over any non-adaptive design.

- **gap**

  Gap is a steady benchmark that does not suffer from performance bottlenecks. The mispredicted branch cost is relatively higher than the others. The AVP chooses branch predictor functionality. For non-adaptive designs, data prefetching seems to be the best choice for gap.

- **gcc**

  Gcc is a benchmark that suffers from many performance bottlenecks, including issue queue being full, mispredicted branches, instruction cache misses and data cache misses. Those bottlenecks have close performance bottleneck intensity. The data cache

miss is relatively higher than others. As a result, the AVP chooses data prefetcher functionality most of time.

- **gzip**

  Gzip suffers mainly from branch misprediction performance bottlenecks. The AVP chooses branch predictor functionality most of time. Besides branch misprediction performance bottlenecks, it incurs data cache misses and issue queue being full bottlenecks as well. The AVP gets additional performance gains with data prefetching and value predictor, while it retains branch predictor's benefit.

- **mcf**

  Mcf executes in two stages. In the beginning of execution, it runs smoothly without any outstanding performance bottlenecks. The issue queue cost is relatively higher, the AVP is in value predictor state during that period. After that, it enters memory intensive code region. Data cache cost becomes the dominant one. AVP is in data prefetcher state then. The AVP gets almost the same result as non-adaptive data prefetcher.

- **parser**

  Performance bottlenecks vary during the parser benchmark execution between mispredicted branch, data cache misses and issue queue being full. The AVP changes its state between branch predictor, data prefetcher and traditional value predictor correspondingly. However, the performance bottleneck change is very frequent (i.e. each performance bottleneck is not very stable). As a result, the AVP does not get better performance than the non-adaptive value predictor.

- **perl**

  Perl mainly suffers from the issue queue being full, instruction cache misses and branch misprediction bottlenecks. The AVP changes its state between traditional value predictor, instruction prefetcher and branch predictor. The AVP outperforms any non-adaptive design.

- **twolf**

  Twolf suffers from issue queue being full, mispredicted branch, data cache misses and instruction cache misses bottlenecks. The data cache misses have relatively higher cost than others, and the AVP chooses data prefetcher functionality most of the execution time.

- **vortex**

  Vortex executes in two stages. In the first half of execution, it suffers from a lot of instruction cache misses. In the other half, it runs smoothly. The instruction cache misses have a little higher cost than others during that period. The AVP chooses instruction prefetcher functionality during the whole execution.

- **vpr**

  Vpr suffers from the issue queue being full, data cache misses and branch misprediction bottlenecks during its execution. The AVP changes its state between traditional value predictor, instruction prefetcher and branch predictor functionalities correspondingly. We can see the AVP outperforms any non-adaptive design.

8.1.2 AVP behavior

Figure 8.3 shows the AVP changes its state during program execution by dividing the whole execution time by different state categories. Unlike the static statistics of state percentage in Figure 8.3, this section presents the AVP on a timeline to display its dynamic behavior.

To demonstrate the AVP dynamic behavior, we choose two different benchmarks, mcf - a memory intensive benchmark, and bzip2 - a mixed bottleneck benchmark. For each graph, there are three charts. The X axis in all of them is the timeline shown as the sequence number of each sampling period. The top chart shows the AVP state behavior. Its Y axis has four states - 0 for traditional value predictor, 1 for branch predictor, 2 for data prefetcher and 3 for instruction prefetcher. The middle chart shows the cost value behavior of data cache misses. The bottom chart shows the cost value behavior of branch mispredictions. The dark points in the cost chart represent the cost value with the AVP. The gray points in the cost chart represent the cost value without the AVP.

Figure 8.4 shows Mcf's behavior. We can see the AVP is in value predictor state from the beginning. During that period, mcf does not have any memory cost and has a small amount of branch misprediction cost. After that, AVP changes to data prefetcher state until the end. In the memory cost chart, AVP significantly reduce the memory cost compared to the original memory cost. There is not much difference in branch misprediction costs, because AVP is not in branch predictor state. Figure 8.5 shows bzip2's behavior. AVP has intermittent data prefetcher and branch predictor state in the beginning part and ending part. In the middle of execution, it is in value predictor state without much data cache cost

and branch misprediction cost. We can see the AVP reduces both data cache miss cost and branch misprediction cost when it is in data prefetcher and branch predictor states.

8.2 Comparison of adaptation algorithms

Figure 8.6 shows the results of the three adaptation algorithms - greedy, history and conservative with default configurations. We can see in most of the benchmarks these three algorithms attain similar performance. However, when we exploit the design space of history algorithm and conservative algorithm, their performance impact changes significantly. We will discuss this in detail in the following two sections.

8.2.1 History vs. Greedy algorithms

Compared to the greedy algorithm, the history algorithm changes the AVP's state only when it sees repeating bottlenecks. The idea of history algorithm is avoiding the "noise bottleneck" transitions. To represent the degree of historyness of the algorithm, we define a variable, called `history threshold`, which represents the minimum number of repeating states that cause a change in the AVP's state. In the design space of history thresholds, there are two extreme ends. One is the zero value. In this case, the history algorithm turns to the greedy algorithm, because it does not need the bottleneck repeat in order to change the AVP's state. On the other extreme end, the history threshold value is infinite. In this case, the history algorithm turns the AVP into a traditional non-adaptive value predictor, assuming that the AVP's initial state is a traditional value predictor.

Figure 8.7 shows the average IPC over the history threshold design space. We

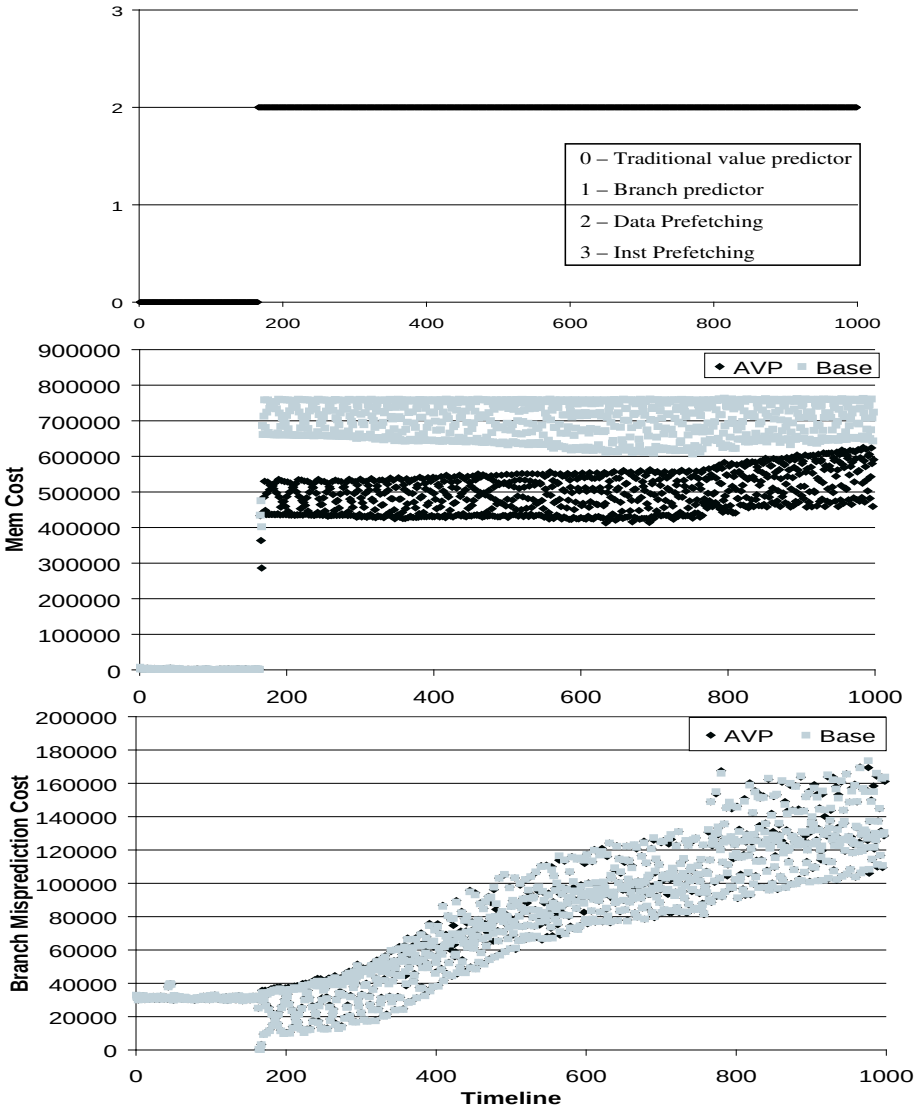

Figure 8.4: AVP mcf behavior with the greedy algorithm

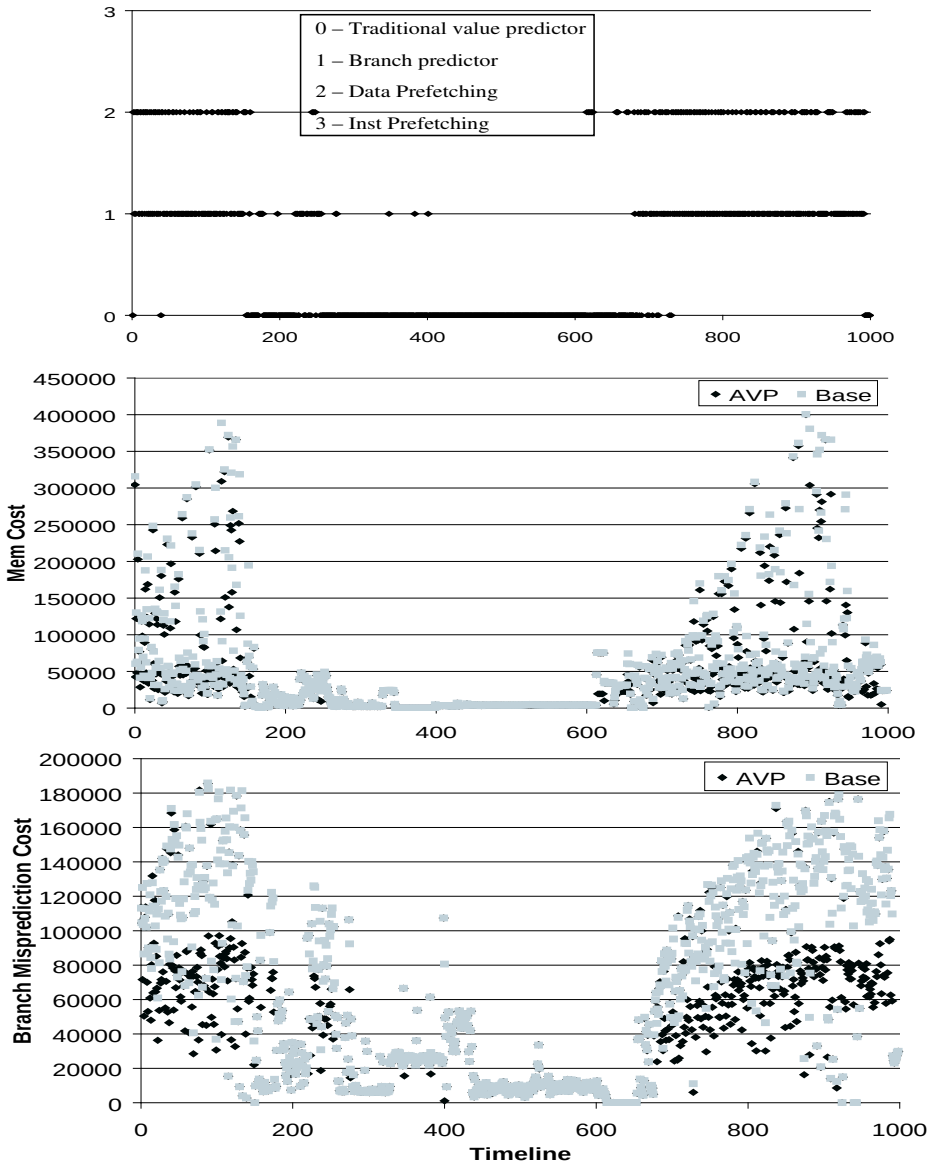

Figure 8.5: AVP bzip2 behavior with the greedy algorithm

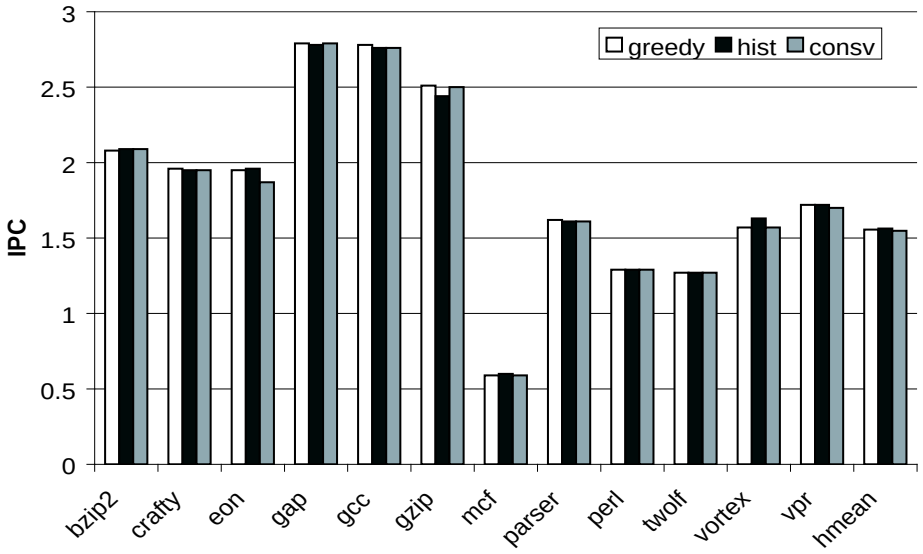

Figure 8.6: Comparison of the three adaptation algorithms

can see when history threshold is a number less than 4, the history algorithm achieves a little better performance than the greedy algorithm (history threshold == 0). After that, as history threshold increases, performance starts decreasing, until we reach the infinite end, where AVP becomes non-adaptive traditional value predictor. Based on Figure 8.7, we conclude that the history algorithm outperforms the greedy algorithm when in the beginning range of design space, when it avoids "noise bottlenecks" while keeping relatively fast response. When the history threshold becomes higher, its response become slows , and that hurts performance.

8.2.2 conservative vs. Greedy algorithms

Compared to the greedy algorithm, the conservative algorithm changes the AVP's state only when the difference between the cost value of the new bottleneck and the cost

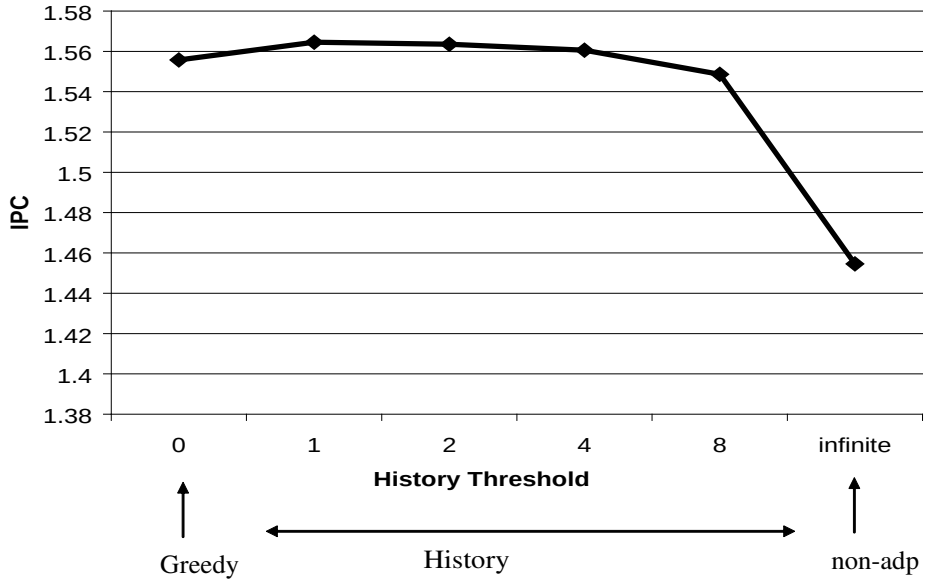

Figure 8.7: History vs. Greedy adaptation algorithms

value of the old bottleneck is greater than a threshold. The idea of the conservative algorithm is avoiding the "state oscillation" by considering the potential performance loss when AVP changes state. To represent the degree of inertia we need to overcome, we define a variable, called `conservative threshold`, which represents the threshold value above which the AVP changes state. Similar to the history algorithm, there are two extreme ends in the design space of smart threshold. One is the zero value. In this case, the conservative algorithm becomes the greedy algorithm, because there is no threshold to control the state change. On the other end, the smart threshold value is infinite. In this case, the conservative algorithm turns AVP into a traditional non-adaptive value predictor, assuming the AVP's initial state is a traditional value predictor.

Figure 8.7 shows the average IPC over the conservative threshold design space. We can see that the conservative algorithm outperforms the non-adaptive design (threshold

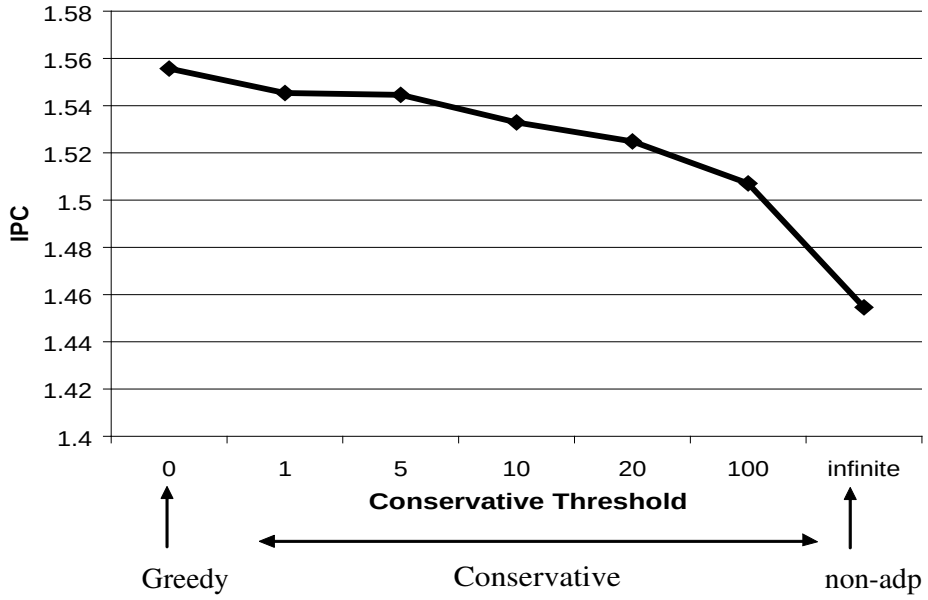

Figure 8.8: Conservative vs. Greedy adaptation algorithms

== infinite). However, it can not beat the greedy algorithm. Considering the motivation of the conservative algorithm, we can conclude that "state oscillation" is not a problem for the SPECK 2K INT benchmarks. The state transition threshold reduces the response time, and that hurts performance.

8.3 Sensitivity study

This section presents an AVP sensitivity study to display the performance variance when certain design parameters change.

8.3.1 Sampling interval sensitivity

There is a tradeoff between performance and sampling frequency. Finer grain sampling periods result in more accurate and faster adaptation. However, state transition

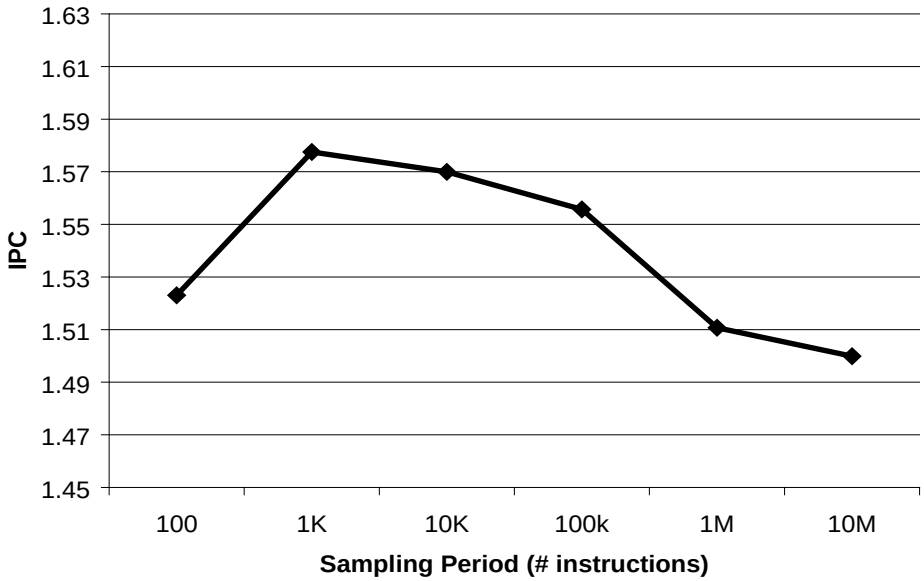

Figure 8.9: Sampling interval sensitivity

is not free. The value predictor table needs training to predict accurately. Figure 8.9 shows the performance based on different sampling intervals. We can see that 1K and 10K sampling intervals give the best results. When the interval is 100 instructions, the states are not stable. As a result, the frequent state transitions hurt performance. When interval increases to 1M and 100M, it reduces the AVP's responsiveness and impact performance negatively.

8.3.2 Value predictor size sensitivity

The major hardware cost of a value predictor is the prediction table. Typically, the bigger the table, the higher the prediction accuracy. When the table is small, there is a lot of aliasing (i.e. different index information is mapped to the same entry of a value predictor table) that results in low prediction accuracy. As the table size increases, the

aliasing pressure is relieved. When the size reaches a point, adding more will not give more performance gains. This saturation point really depends on the design of the value predictor, mostly depends on the index scheme. In our work, we use PC indexed two-delta value predictor. Each entry consists of tag (the rest of PC without index bits), the value, the stride, and the confidence state. So for each entry, the size is about 12 bytes. The index information is instruction PC. The lower bits are used to index into the corresponding entry. The higher bits form the tag to identify hits or misses in the prediction table.

Figure 8.10 shows the performance change when the table size varies. The X axis represents the number of prediction table entries. The Y axis represent the average IPC. We can see after 1K size, the performance does not get much improvement. That is because PC is the index information and for most of the benchmarks the instruction working set is limited. For example, mcf's major working set is while loop and a small amount of instructions reside in it. And also, our value predictor is stride-based. The amount of stide pattern in program is limited as well.

8.3.3 Cost unit sensitivity

There are two reasons we introduce the cost unit concept. One is, as we discussed in Chapter 5, cost unit can reduce the number of bottleneck phases in order to increase the predictability of performance bottlenecks. The other is that the cost unit can group similar cost values into the same cost level and increase the stability of the AVP. We do not want to change the AVP's state when the difference in raw bottleneck costs is very small, because AVP state transition is not free. Certainly, there is trade-off in the choice of the cost unit value. A high cost unit will reduce the AVP's sensitivity and miss performance bottleneck

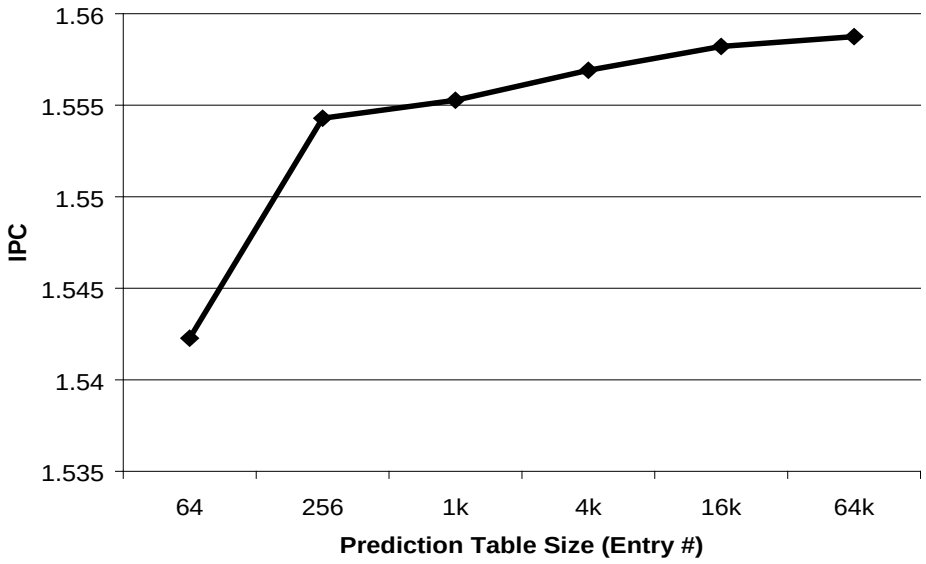

Figure 8.10: Hardware budget sensitivity

changes.

Figure 8.11 shows the results of the cost unit sensitivity study. We can see when the cost unit increases but remains less than 10, the performance gets a little improvement. After 10, the performance decreases significantly, as the cost unit increases.

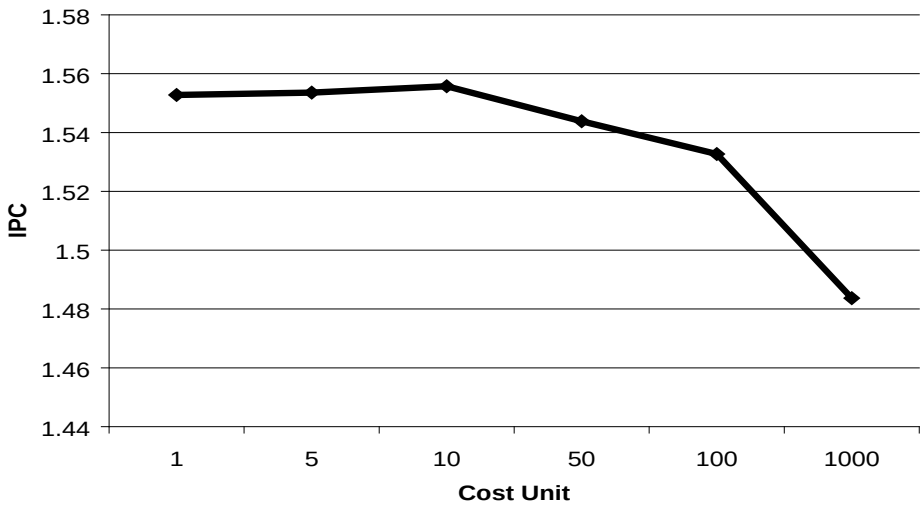

Figure 8.11: Cost accuracy sensitivity

Chapter 9

Conclusion

As circuit technology and microarchitectures develop, modern microprocessors are getting more on-chip transistors and its microarchitecture is getting more complex to extract more ILP/TLP. The increasing hardware budget leads to more power consumption which results in more design problems, because energy efficiency is becoming an important criteria for modern microprocessors. Meanwhile, the shrinking feature sizes make the power and heat density to increase steadily, and that creates vast difficulties in reliability and cooling costs. The design of a modern microprocessor becomes a procedure involving all performance, power and thermal concerns.

It has been shown that different applications have different characteristics and even the same application exhibits different behaviors during its execution. The variation of applications and portions of an application results in different requirements and utilization for system resources. Adaptive microarchitectures, as a runtime optimization, build flexibility into the processor. It has become a topic of great interest in the computer ar-

chitecture community, because it can change its configuration at runtime according to the application needs. Because of the increasing complexity of modern microarchitecture and number of design concerns, there are an increasing number of on-chip optimization choices. We argue that system-wide optimization is the future of adaptive microarchitectures.

In this work, we propose a micro-architectural model - CiC to implement the system-wide optimization scheme. The key point of the CiC model is guiding runtime optimizations with the consideration of overall system performance and different optimization tradeoffs by using system-wide information instead of local information, to maximize the benefit of adaptive resources. There are two fundamental functions related to CiC design - distributed system information collecting and centralized information analysis. Corresponding to these two functions, there are two logic components inside CiC - a monitor and a controller. The monitor is in charge of collecting system-wide information and the controller is for information processing.

To demonstrate the CiC model, we present a performance-oriented adaptive microarchitecture design - adaptive value predictor, which has four functionalities (traditional value predictor, branch predictor, instruction prefetcher and data prefetcher). It adapts its functionalities to performance bottlenecks to maximize the effectiveness of the value predictor. The most important technical issue for adaptive value predictor is how to identify performance bottlenecks at runtime. We propose a low-cost counter-based performance bottleneck model, which quantifies the performance bottlenecks with a vector, called the bottleneck vector. Based on the bottleneck vector, we can identify and predict runtime performance bottlenecks easily. Unlike prior program phase work, which search the design

space to find optimal adaptation choices, our scheme can pinpoint where the performance limitations are and make adaptation decisions accordingly. Using our performance bottleneck analysis model, we design the CiC monitor of the AVP that is in charge of collecting related system wide performance information. We also design the CiC controller of the AVP to implement an adaptation algorithm. There are three adaptation algorithms studied in our work: greedy, history and conservative. The greedy algorithm tries to respond to performance bottleneck changes as soon as possible. In contrast, the history algorithm tries to filter the noise phases and conservative algorithm tries to avoid state oscillation. Our results show that AVP can outperform baseline design by 30% and non-adaptive value predictor by 10% on average.

For CiC design concept, there are several future steps we can take.

- **Performance Bottleneck Guided Power Management**

  For power management schemes, there are always performance and power dissipation tradeoffs. If system-wide performance bottleneck information is available, it can guide power management in a more efficient way. A simple management scheme could be always giving non-bottleneck components a higher priority to enter into power-saving mode to minimize the performance loss.

- **Performance Aware Thermal Management**

  It has been shown that there are hotspots on a chip that may cause system failure. System performance analysis can help thermal management to minimize the performance loss when triggering a thermal management operation. When the hot component is a non-bottleneck component, we can directly slowdown this component

as typical Dynamic Thermal Management does. When the hot component is bottleneck component, in order to minimize performance loss, the first choice to relieve the temperature of the hotspot is not to slow it down directly. Since heat conducts to the surrounding area of the hotspot, we can reduce the surrounding temperature to relieve hotspot indirectly.

- **SMT resource management**

  Simultaneous MultiThreading (SMT) improves overall instruction throughput by simultaneous sharing of microprocessor resources among multiple threads [52, 15]. Therefore, the key to high performance of SMT is partitioning shared resources judiciously among active threads. There are many resource management schemes to optimize the sharing of some microprocessor resources. The typical shared resources are issue queue, reorder buffer and bandwidth. One potential application of CiC model is managing the partition of multiple resource with system-wide performance analysis.

- **Adaptive prime number indexing for shared caches**

  In a Chip Multi-Processor (CMP) architecture, the L2 cache is typically shared by multiple processors to maximize resource utilization and avoid costly resource duplication. However, resource sharing leads to cache contention that impacts performance negatively. One way to relieve cache contention is partitioning the shared cache into different chunks for each thread according to their needs. The resulting problem for this partition is increasing the conflict possibility among accesses from same thread. Kharbutli et al. [30] propose a prime-number based indexing scheme, which can dis-

tribute data allocation evenly among different sets. The choice of prime number depends on the number of sets. One adaptive design for a shared cache is dynamically partitioning cache into separate chunks for each application, and then adapting the indexing prime number accordingly.

Overall, the CiC design model is a generic micro-architectural model for system-wide optimization. The AVP, as an application of CiC model, is proven to be an effective adaptive design by tuning it functionalities to runtime performance bottlenecks.

Bibliography

[1] D.H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, Dec. 1999.

[2] R. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proc. of the 28th Intl. Sym. on Computer Architecture*, July 2001.

[3] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *Proc. of the 33rd Annual Intl. Sym. on Microarchitecture*, Dec 2000.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 26–37, 2001.

[5] E. A. Brewer, C. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Measurement and Modeling of Computer Systems*, pages 247–248, 1992.

[6] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[7] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[8] A. Buyuktosunoglu, S. Schuster, P.Cook D. Brooks, P. Bose, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems (PACS2000, held in conjunction with ASPLOS-IX)*, July 2000.

[9] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in smt processors. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 171–182, 2004.

[10] J. Chang and M. Pedram. Energy minimization using multiple supply voltages. In *International Symposium on Low Power Electronics and Design*, 1996.

[11] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[12] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[13] A. S. Dhodapkar and J. E. Smith. Managing multiconfiguration hardware via dynamic working set analysis. In *Proc. of the 29th Annual Intl. Sym. on Computer Architecture*, May 2002.

[14] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*, December 2003.

[15] G. K. Dorai and D. Yeung. Transparent threads: Resource sharing in smt processors for high single-thread performance. In *11th International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[16] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.

[17] B. A. Fields, R. Bodk, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Micro-36*, 2003.

[18] 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.

[19] D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Workshop on Complexity- Effective Design (WCED2000, held in conjunction with ISCA27)*, Jun 2000.

[20] S. Ghiasi, J. Casmira, and D. Grunwald. Using ipc variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design*, July 2000.

[21] A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. In *IEEE Transactions on Parallel and Distributed Systems*, pages 28–40, 1993.

[22] J. Gonzalez and A. Gonzalez. Control-flow speculation through value prediction for superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[23] S. Haga, Y. Zhang, and R. Barua. Execution history guided instruction prefetching. In *Proc. Intl. Conf. on Supercomputing*, 2002.

[24] W.-C. Hsu and J. E. Smith. A performance study of instruction cache prefetching methods. In *IEEE Transactions on Computers*, 1998.

[25] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Int. Symp. Low Power Electronics and Design*, 1998.

[26] T. L. Johnson and W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 315–326, 1997.

[27] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[28] T. Juan, S. Sanjeevan, and J. Navarro. Dynamic history- length fitting: A third level of adaptivity for branch prediction. In *Proc. of the 25 t Intl. Sym. on Computer Architecture,*, pages 155–166, July 1998.

[29] S. Kaxiras, Z. Hu, and M. MartonosiT. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th annual international symposium on Computer architecture*, 2001.

[30] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *10th International Symposium on High Performance Computer Architecture*, 2004.

[31] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.

[32] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *International Symposium on Performance Analysis of Systems and Software*, Mar 2004.

[33] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[34] C.-H. Lim, W. Daasch, and G. Cai. A thermal-aware superscalar microprocessor. In *International Symposium on Quality of Electronic Design*, 2002.

[35] M.H. Lipasti and J.P. Shen. Exceseding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[36] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[37] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking*, 2001.

[38] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: Demand based associativity via global replacement. In *Proceedings of the 32th annual international symposium on Computer architecture*, pages 544–555, 2005.

[39] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[40] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[41] G. Semeraro. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO35*, 2002.

[42] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *International Symposium on High-Performance Computer Architecture*, 2002.

[43] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct 2004.

[44] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[45] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the 30th Annual Intl. Sym. on Computer Architecture*, Jun 2003.

[46] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, June 200.

[47] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proc. Intl. Conf. on Supercomputing*, 1992.

[48] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32th annual international symposium on Computer architecture*, pages 346–356, 2005.

[49] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *International Symposium on High-Performance Computer Architecture*, 2005.

[50] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro*, Aug 2002.

[51] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[52] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23th International Symposium on Computer Architecture*, 1996.

[53] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X.Ji. Adapting cache line size to application behavior. In *Intl. Conf on Supercomputing*, pages 145–154, July 1999.

[54] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[55] M Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Supercomputing*, November 1996.

[56] H. Zhou, J. Flanagan, and T. M. Conte. Detecting global stride locality in value streams. In *30th International Symposium on Computer Architecture*, 2003.