Energy-Efficient Acceleration of Asynchronous Programs

by

Gaurav Chadha

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

Doctoral Committee:

Associate Professor Satish Narayanasamy, Co-Chair
Professor Scott A. Mahlke, Co-Chair
Associate Professor Thomas F. Wenisch
Associate Professor Zhengya Zhang

© Gaurav Chadha 2016

All Rights Reserved

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Asynchronous or event-driven programming has become the dominant programming model, with it now being used to develop a wide range of systems, including mobile and Web 2.0 applications, Internet-of-Things, and even distributed servers. In this programming model, computation tasks are posted as events to an event-queue, from where they are processed asynchronously. Due to the rich array of asynchronous inputs to the above systems, the asynchronous programming model is a natural fit.

Execution characteristics of asynchronous programs significantly differ from synchronous programs as they interleave short events from varied tasks in a fine-grained manner. We observe that, despite their ubiquitous use, these programs perform poorly on conventional processor architectures that are heavily optimized for the characteristics of synchronous programs.

The processor industry has embraced heterogeneous multi-processor designs with domain-specific accelerators as an answer to the end of Dennard Scaling. We envision a system where at least some of these cores could be customized to efficiently execute asynchronous or event-driven programs. To this end, this thesis proposes novel processor optimizations for accelerating asynchronous programs.

Using Web 2.0 applications as a case in study, we found out that these suffer from very poor cache and branch predictor performance. To alleviate these performance bottlenecks,

first, we describe an instruction prefetcher, EFetch, designed for asynchronous programs. We found that an *event signature*, which captures the current event and function calling context, is a good predictor of the control flow inside a function of an event-driven program, facilitating accurate instruction prefetch. For a set of real-world popular asynchronous Web 2.0 applications including Amazon, Google maps, and Facebook, we show that EFetch outperforms the commonly implemented next-2-line prefetcher by 17%.

The second optimization, the Event Sneak Peek (ESP) architecture, exploits the fact that events are posted to an event queue before they get executed. By exposing this event queue to the processor, ESP gains knowledge of the future events. Instead of stalling on long latency cache misses, ESP jumps ahead to pre-execute future events and gathers useful information that later help initiate accurate instruction and data prefetches and correct branch mispredictions. For the same set of Web 2.0 applications as above, ESP improves performance by an average of 19%.

The third optimization, the Eidetic Event Rehearsal (EER) system, exploits the recurrence of events within and across browsing sessions to accelerate events while saving energy. It speculatively pre-executes an event a few times, and thereafter uses the pre-recorded information on future instances of the same event without speculative pre-execution, thus saving energy. In this way it exploits the latent event-level parallelism (ELP) present in these applications similar to ESP, but does so using an energy-efficient software-centric design that requires minimal hardware support. For a similar set of Web 2.0 applications as above, EER improves performance by an average of 43% while saving 18% energy.

Our research suggests that, with the ubiquitous use of the asynchronous programming model, it is worthwhile customizing the processor to improve the performance of asyn-

chronous programs.

CHAPTER I

Introduction

Asynchronous or event-driven programming [5] has become the dominant programming model in the last few years. In this model, computations are posted as events to an event queue from where they get processed asynchronously by the application. A huge fraction of computing systems (Figure 1.1) built today use asynchronous programming. All the Web 2.0 JavaScript applications (e.g., Gmail, Facebook) use asynchronous programming. There are now more than two million mobile applications available between the Apple App Store and Google Play, which are all written using asynchronous programming. Distributed servers (e.g., Twitter, LinkedIn, PayPal) built using actor-based languages (e.g.,

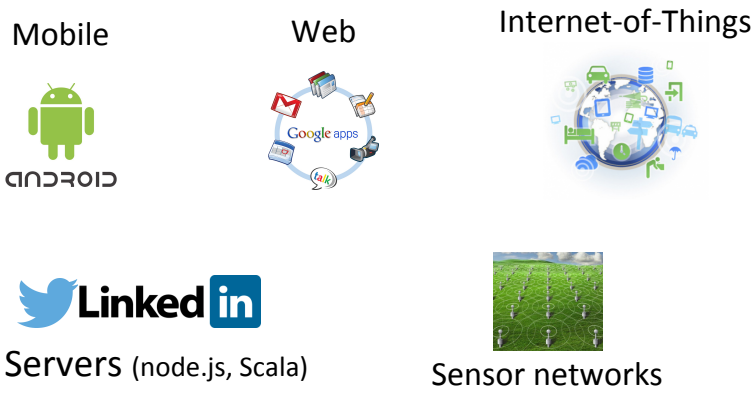


Figure 1.1: Ubiquitous use of asynchronous programming.

Scala) and platforms such as `node.js` rely on asynchronous events for scalable communication. Internet-of-Things (IoT), embedded systems, sensor networks, desktop GUI applications, etc., all rely on the asynchronous programming model.

We consider the Web 2.0 applications as a case in study. Web applications are increasingly popular today as it allows users to easily run them within their web browsers, offering portability, and access data stored in the cloud. These web applications, written in a scripting language (mostly JavaScript), receive asynchronous input from diverse input sources such as user clicks, servers over the network, microphone, camera and other sensors. Web development platforms use the asynchronous or event-driven programming model as it is a natural fit for handling this rich array of asynchronous input.

1.1 Asynchronous Program Performance: Challenges

In spite of the ubiquitous use of asynchronous programming, today's general-purpose processor architectures are heavily optimized for the synchronous programming model, and ignores the unique execution characteristics of asynchronous programs.

Asynchronous programs present unique challenges and opportunities for performance optimization. Unlike a synchronous program that executes one long (billions of instructions) task after another, an asynchronous program's execution consists of many short (only millions of instructions) events orchestrated by the timing of external events (e.g., user click). Such fine grained interleaving of many small and varied tasks destroys instruction and data locality, and exposes little repeatable patterns assumed by processor components such as the branch predictor. Cold misses, which are of little concern in synchronous pro-

grams, constitute a significant source of pipeline stalls in asynchronous programs with short events. In this work, we recognize the following challenges and opportunities, and propose solutions.

Asynchronous programs, especially web applications, tend to exercise large instruction footprints to support a rich set of features, resulting in significantly higher instruction cache miss rates than synchronous programs. L1 instruction cache misses significantly degrade performance. On average, across the web applications studied, performance can be increased by 53% if all L1-I cache misses are eliminated. We propose an instruction prefetcher customized for asynchronous programs to help alleviate the performance penalty due to L1-I cache misses (Chapter III).

Despite the performance challenges of asynchronous programs, they also present unique opportunities. In an asynchronous program, events are enqueued in an "event queue" before they are processed sequentially by a thread. By exposing the event queue to the processor, the processor can gather information about them ahead of time. This information can then be used to significantly improve the accuracy of various hardware predictors when an event is executed. We have designed an architecture that takes advantage of the above insight to accelerate events' execution (Chapter IV).

Exploiting the latent event-level parallelism (ELP) present in these applications in a similar manner as above, this thesis proposes an energy-efficient software-centric design to accelerate events while saving energy in Chapter V. With minimal hardware support, this design significantly accelerates events while saving energy, which has become a first-order design consideration with wide proliferation of mobile devices.

1.2 Processor Optimizations

As the processor industry continues to embrace heterogeneous processor designs with domain-specific accelerators as an answer to the end of Dennard scaling to mitigate this "dark silicon" problem [24], we envision that at least a few of the processor cores could be customized for efficiently executing the asynchronous programs. To this end, this thesis proposes novel processor optimizations for accelerating asynchronous programs.

1.2.1 Accelerating Instruction Fetch

L1 instruction cache misses significantly degrade performance. On average, across the web applications studied, performance can be increased by 53% if all L1-I cache misses are eliminated.Unlike data cache misses, out-of-order execution cannot hide the latency of an instruction cache miss that stalls the pipeline. While there is a rich literature on designing instruction prefetchers to address this problem [30, 42, 57, 48], past prefetcher designs have significant limitations when applied to event-driven programs. They either rely on spatial locality [30, 48], address access patterns [57], predictability of branches [15], or require fairly large hardware structures [25]. In asynchronous programs, in an otherwise seemingly continuous instruction stream, different events execute distinct code supporting a diverse set of functionalities. This makes it very hard for these conventional instruction prefetchers to perform well. The insight here is to use an event identifier (event-ID) to separate out distinct events, and then look for patterns within an event and across invocations of the same event.

To mitigate the performance penalty incurred due to high instruction cache miss rates,

this thesis proposes **E**vent **Fetch**, or EFetch, an instruction prefetcher customized for asynchronous programs. EFetch exploits the above insight, and further uses the function call context to accurately predict the control flow inside a function, leading to accurate instruction prefetches.

1.2.2 Exploiting Event-Level Parallelism

In an asynchronous program, events are enqueued in an "event queue" before they are processed sequentially by a thread. By exposing the event queue to the processor, the processor can know the sequence of events that are going to be executed in the future. Our insight is that this knowledge can be exploited to take a "Sneak Peek" into the executions of the future events and gather information about them ahead of time. This information can then be used to significantly improve the accuracy of various hardware predictors when an event is executed.

This thesis proposes the **E**vent **S**neak **P**eek (ESP) architecture that exploits the above insight to significantly reduce instruction and data cache misses, and branch mispredictions for asynchronous programs. On encountering a long latency last-level cache (LLC) miss for instruction or data, ESP uses the idle CPU cycles to "jump ahead" to the next event in the event queue and speculatively pre-execute it. While pre-executing an event, ESP records the sequence of instruction and data cache blocks accessed, and also the branch mispredictions. ESP jumps back to the normal execution once the LLC miss gets resolved. Later, during the normal execution of a pre-executed event, the previously recorded information is used to initiate accurate prefetches and correct branch mispredictions.

Pre-execution of future events is speculative as they may depend on the earlier skipped

events. However, since the events perform varied tasks, they are fairly independent of each other. Thus, speculative pre-execution closely matches the normal execution and helps collect accurate predictions. ESP does not use the computed results from the speculative pre-execution of event, as it may require fairly complex hardware such as the components used in Thread-Level Speculation (TLS) [27, 35, 49, 52] to check for dependencies between events and recover from mis-speculations.

While ESP achieves significant performance improvement it incurs an energy and hardware storage overhead. **E**idetic **E**vent **R**ehearsal (EER) exploits the latent event-level parallelism present in these asynchronous applications similar to ESP, however it uses an energy-efficient software-centric approach. Considering Web 2.0 applications, an event executes repeatedly in a web browsing session and across browsing sessions. EER exploits this recurrence of events to speculatively pre-execute each event a few times and thereafter, use the pre-recorded information each time the event repeats without repeated and redundant pre-executions. This makes EER highly energy-efficient.

EER has been designed primarily in software with minimal hardware support, and no hardware storage overhead. With virtual memory as the backing store EER can afford to record information about a large number of events and also use it across browsing sessions. Moreover, it can be deployed on multiple different underlying hardware substrates like SMT and CMP.

1.3 Organization

The rest of the document is organized as follows:

Chapter II presents useful background material on the asynchronous programming model, how it is viewed by conventional processor architectures, and the web browser software architecture.

Chapter III presents EFetch, an instruction prefetcher designed for asynchronous programs.

Chapter IV describes the ESP architecture, which further exploits the unique optimization opportunities offered by the asynchronous programming model, to help enhance the performance of caches and the branch predictor.

Chapter V discusses an energy-efficient software-centric approach for accelerating asynchronous programs called EER.

Chapter VI finally concludes this thesis.

CHAPTER II

Background

This chapter presents background material useful in understanding the subsequent chapters. I start with describing the basic functioning of the asynchronous programming model. The next section, points out an important difference in how the execution of an asynchronous program is viewed by conventional processor architectures as opposed to the execution of synchronous programs. Since we use Web 2.0 applications executing in a web browser as a case in study, it is useful to understand the web browser software architecture. The last section describes it, and puts into focus the part of the web browser targeted in our study.

2.1 Illustration: Asynchronous Web Application's Execution

Figure 2.1 shows an example execution of an asynchronous web application. The events to be executed are held in an event-queue. A looper thread in the renderer process of a web browser, constantly polls an event-queue. It dequeues one event at a time and invokes the necessary JavaScript (JS) handler function to process the event. The events may be

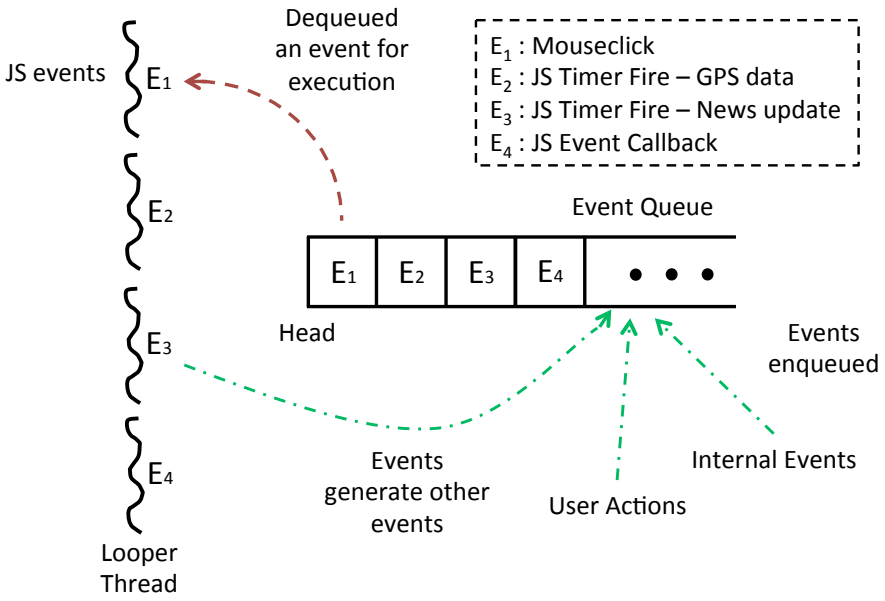


Figure 2.1: Asynchronous execution of a web application.

generated internally or in response to an external input. If an event has to wait for any long-latency operation (e.g., downloading a web page), instead of stalling, the event handler would register an event callback to be invoked when the long-latency operation has completed, and then return.

2.2 Asynchronous Vs Synchronous Programming Models

For the purpose of illustration, let us consider a program consisting of three distinct tasks shown in Figure 2.2. A typical single-threaded synchronous program completely finishes a task before starting another. In a multi-threaded synchronous program, the tasks can be assigned to different threads. These threads may either run concurrently on different processor cores as shown in Figure 2.2, or the OS may interleave some of them at a coarse granularity on a single core. In both models, as the same task is executed for long periods of time (typically billions of instructions), they exhibit a high degree of locality and repetitive

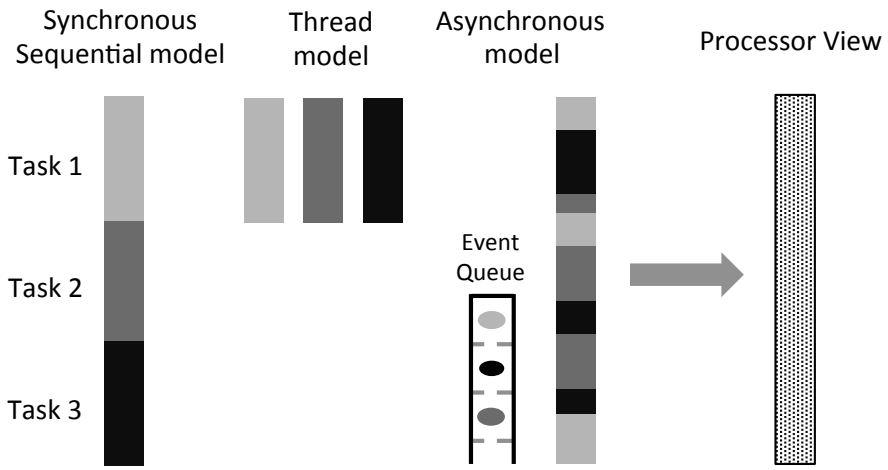


Figure 2.2: Comparison of Programming models. A processor's simplistic view of an asynchronous program's execution.

behavior, which current processors exploit to achieve high cache performance and branch prediction accuracy.

In computing systems with frequent long latency blocking operations (mostly due to I/O), synchronous programs waste processor resources waiting for those operations to complete. An asynchronous program avoids this problem by splitting a task into multiple sub-tasks at the boundaries of blocking operations and associates them with event conditions. When the required event condition is met (e.g., user click), the corresponding sub-task (event handler) is enqueued into an event queue. Instead of blocking on an I/O operation, an asynchronous program switches to execute a ready sub-task from the event queue. Programmers also take care to ensure that each event execution is short, because otherwise the system may become unresponsive.

Such a fine-grained interleaving of short (only millions of instructions) sub-tasks (events) from different tasks (Figure 2.2) destroys instruction and data cache locality in asynchronous programs. Also, it exposes little repetitive behavior, which is critical for achieving prediction accuracy. Unfortunately, conventional processors are presented with a simplistic

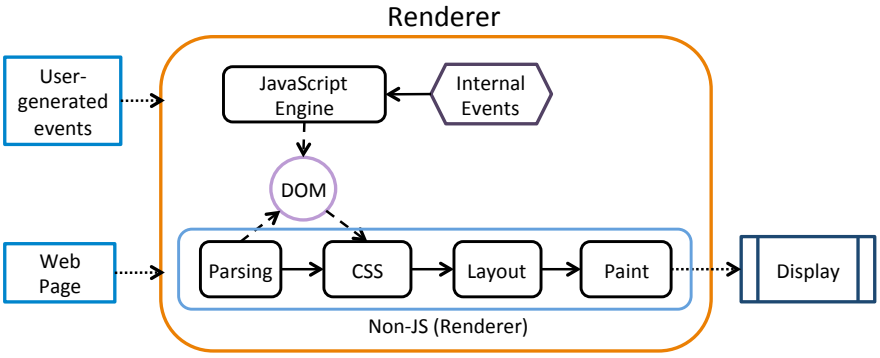


Figure 2.3: Software components of a renderer process in a browser.

homogeneous view of an asynchronous thread (Figure 2.2), which is far removed from reality.

2.3 Web Browser Software Architecture

A web browser consists of several processes. For each user session ("tab"), the browser spawns off a renderer process, which performs the critical tasks of a browser. A main process in the web browser receives events from the external system and delivers them to the appropriate renderer process.

The software components of a renderer process are shown in Figure 2.3. They are responsible for tasks such as parsing `HTML` content, layout, and paint. When it encounters events that need to be processed by executing a JavaScript (JS) program, it invokes the JavaScript engine. The JavaScript engine starts executing the source code in the interpreted mode first (or a very basic compiled code), and then dynamically compiles hot functions into native code.

CHAPTER III

EFetch: Optimizing instruction fetch for asynchronous Web 2.0 applications

Web 2.0 applications written in JavaScript are increasingly popular as they are easy to use, easy to update and maintain, and portable across a wide variety of computing platforms. Web applications receive frequent input from a rich array of sensors, network, and user input modalities. To handle the resulting asynchrony due to these inputs, web applications are developed using an asynchronous or event-driven programming model. These event-driven web applications have dramatically different characteristics, which provides an opportunity to create a customized processor core to improve the responsiveness of web applications.

In this chapter, we take one step towards creating a core customized to event-driven applications. We observed that instruction cache misses of web applications are substantially higher than conventional server and desktop workloads due to large working sets caused by distant re-use. To mitigate this bottleneck, we propose an instruction prefetcher (EFetch) that is tuned to exploit the characteristics of web applications. We found that an *event sig-*

nature, which captures the current event and function calling context, is a good predictor of the control flow inside a function of an event-driven program. It allows us to accurately predict a function's callees and their function bodies and prefetch them in a timely manner. For a set of real-world web applications, we show that the proposed prefetcher outperforms commonly implemented next-2-line prefetcher by 17%. Also, it consumes 5.2 times less area than a recently proposed prefetcher, while outperforming it.

3.1 Introduction

Web 2.0 has revolutionized the way we use personal computers today. Modern websites are extremely dynamic, feeding personalized contents to the users according to their preferences. Web 2.0 has also enabled a new class of client-side web applications such as web emails, interactive maps, and social networks. A web application is essentially a program that runs within a web browser. Web applications are increasingly popular today due to the ease with which users can run these applications within their web browsers without having to download and install them on their personal computers. Web applications also allow developers to instantly update and maintain them. Furthermore, as they can run within almost any web browser, they are highly portable across diverse systems ranging from servers and desktops to tablets and smart phones.

A large fraction of the Web 2.0 content is being programmed using JavaScript today, as it is commonly supported in most popular web browsers in use. Web applications written in scripting languages receive input from diverse sources such as user clicks, accelerometers, microphones, and other sensors. These rich array of input sources provide large asyn-

chronicity to the input stream of web applications. As a result, *event-driven* programming models naturally arise among popular web development platforms. An event-driven model makes it easy to integrate input from a rich array of sensors and user input modalities.

Event-driven web applications typically execute thousands of events in a second. Runtime characteristics of the event-driven web applications developed using scripts are dramatically different from the conventional server or desktop applications, which have been the primary focus for most processor optimizations studied till today.

As Moore's Law continues to hold true, we may be able to continue to increase the number of processor cores in a chip. However, due to the end of Dennard scaling, it is also likely that we may not be able to power-up and operate all the processor cores at the same time. Heterogeneous processors are a likely solution to mitigate this "dark silicon" problem [24]. Given that web applications constitute a dominant use for consumer devices, we envision that one or more of the cores in a heterogeneous multi-core processor could be "Web Cores" that are customized for executing web applications.

In this paper, we take the first step towards designing a WebCore by identifying an important performance bottleneck in event-driven web applications, namely, instruction fetch, and propose an instruction prefetcher, **E**vent **Fetch** or EFetch, to mitigate it. We studied several Web 2.0 applications and discovered that L1 instruction cache misses to be a critical performance bottleneck. Instruction fetch is a more severe problem for web applications than it is for certain server applications that have been used in the past instruction fetch optimization studies [25, 26]. While the L1 instruction cache misses per kilo-instructions (mpki) is on the order of 1-3 for conventional applications, it is nearly 25 on average for popular web applications, e.g., Facebook, Gmail, Amazon, CNN, Google maps.

Event-driven web applications experience poor instruction cache performance for several reasons. First, the size of JavaScript code associated with most web sites is very large, ranging from 200 KB to several megabytes [43]. Second, not only is their instruction footprint larger than conventional programs, but they also do not exhibit much temporal locality for instruction addresses. The reason is that, diverse set of events are invoked in response to external inputs, resulting in too many hot functions. Also, each of those events is designed such that it executes for a fairly short period of time (thousands of instructions on average, millions in the worst case) to ensure responsiveness. In a conventional application, iteration counts of hot loops could be in hundreds of thousands, which causes them to exhibit much more temporal locality. But, loops in web applications rarely ever iterate for a long time. Finally, only about a tenth of a function body gets accessed when a function is invoked. This combined with rich control flow within a function, results in poor spatial locality.

L1 instruction cache misses significantly degrade performance. On average, across the web applications studied, performance can be increased by 53% if all L1-I cache misses are eliminated. Unlike data cache misses, out-of-order execution cannot hide the latency of an instruction cache miss that stalls the pipeline. While there is a rich literature on designing instruction prefetchers to address this problem [30, 42, 57, 48], past prefetcher designs have significant limitations when applied to event-driven programs. They either rely on spatial locality [30, 48], address access patterns [57], predictability of branches [15], or require fairly large hardware structures [25].

In this chapter, we define *event signatures* to design a custom prefetcher for web applications. An event signature is a hash of current event-identifier and function call context signature (a hash of call addresses of functions at the top of call stack). The web browser

is responsible for constructing an identifier for an event and initializing a special hardware *event register* before beginning to process an event's handler. We observed that the event signature is a good predictor of the control flow inside a function, which in turn allows us to predict a function's callees and their function bodies.

In addition to the event register, the WebCore architecture has a hardware *callee predictor table* that keeps track of the set of callees invoked from a function under an event signature context. On encountering a previously seen event signature, WebCore issues prefetch requests to its callees. A prefetcher is effective only if it can prefetch the cache blocks ahead of their use. EFetch prefetches the next function to be called, going down the program call graph in a depth-first manner, staying one function ahead of the program execution. A *predictor stack* keeps track of the predicted call graph and validates as the trailing program makes progress. In the case of a mispredicted call, a recovery is initiated to start the prefetch from the correct functional call context.

We studied a representative website from each class of web applications: e-commerce (`amazon`), interactive maps (`google maps`), search (`bing`), social networking (`facebook`), news (`cnn`), utilities (`google docs`), and data-intensive applications (`pixlr`). This chapter shows that WebCore's prefetcher (EFetch) outperforms commercially implemented next-2-line prefetcher by 17%. It also outperforms a recently proposed prefetcher, PIF [25], while consuming 5.2 times less area.

3.2 Background and Motivation

In this section, I briefly describe a web browser system and the event-driven programming model used to write Web 2.0 applications.

3.2.1 Web Browser Software Architecture

A web browser consists of several processes. For each user session ("tab"), the browser spawns off a renderer process, which performs the critical tasks and the large majority of computation of a browser.

The software components of a renderer process are shown in Figure 2.3. The JavaScript (JS) event handlers are generated by the JS engine which is a part of the renderer. A more detailed description appears in Section 2.3.

Web applications are written in JavaScript using an event-driven programming model. Figure 2.1 shows an example execution of a web application. Events are enqueued into an event-queue, from where they are dequeued by a looper thread, one at a time, and are thereafter, processed by invoking the necessary JS handler. Section 2.1 describes this mechanism in more detail.

3.2.2 Motivation for WebCore

Characteristics of event-driven programs are significantly different from the conventional programs. We studied several micro-architectural characteristics of web applications and discovered that instruction cache misses and branch mispredictions are significantly higher than most conventional programs studied in the past.

Web applications suffer from an instruction fetch bottleneck due to the event-driven nature of these programs. These applications execute a wide range of functions in response to different events initiated by the user and the external system. Furthermore, each function in JavaScript executes for only a few hundreds of instructions to ensure responsiveness. As a result, event-driven JavaScript programs exhibit little temporal locality. They also tend to expose relatively less spatial locality due to rich control flow within the body of a function, which is written to handle a variety of control states within the web application. Finally, the instruction footprint of web applications on average is about 200 KB but could be as high as 2 MB (e.g., `google maps`).

As the processor industry moves towards heterogeneous multi-core processors, we envision that one or more cores could be customized to exploit the characteristics of web applications. In such a scenario, the JS component of the web browser would execute on a dedicated processor core ("Web Core") different from the one used for executing the other parts of the renderer process as shown in Figure 3.1a.

Figure 3.1b shows the L1 instruction cache (I-cache) mpki when we execute the JS events along with all the renderer process. On average, L1 I-cache mpki could be as high as 29. I-cache mpki reduces to about 24 when we execute the native code produced by the JS engine on a separate core. Though, by executing the JS events on a separate core, L1 data cache suffers from coherence misses. However, the impact of this is very small, mitigated, in part, by the additional L1 cache capacity on the separate core. There is a loss of 2.1% in performance compared to running the JS events with the renderer process. In our study, we seek to optimize the I-cache performance of a "Web Core" that is dedicated to execute the instructions corresponding to the JS events.

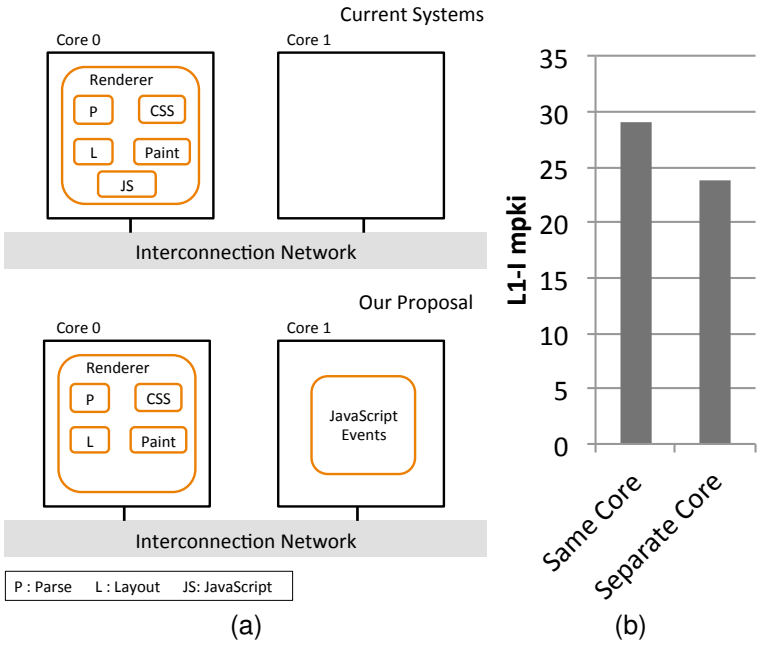


Figure 3.1: (a) Isolating JavaScript Events, (b) L1-I mpki when executing JS events along with the entire Renderer (Same Core) or Isolated (Separate Core)

Figure 3.2 shows the I-cache mpki for the several web applications we studied. For reference, it also shows the average mpki we observed for conventional PARSEC [10] and SPECint 2006 [3] workloads. I-cache mpki for web applications could be as high as 26 (`bing`), where for SPECint 2006 it is about 2.

Figure 3.3 shows the cumulative I-cache L1 miss rates observed due to different cache blocks accessed. As the figure shows, over 10,000 cache blocks are needed to cover 80% of misses for web applications. However, a few hundred cache blocks can cover almost all the I-cache misses for the conventional PARSEC programs, or a couple thousand for SPECint 2006 programs. These results indicate the need for an instruction prefetcher that is customized to exploit the characteristics of the event-driven JavaScript web applications.

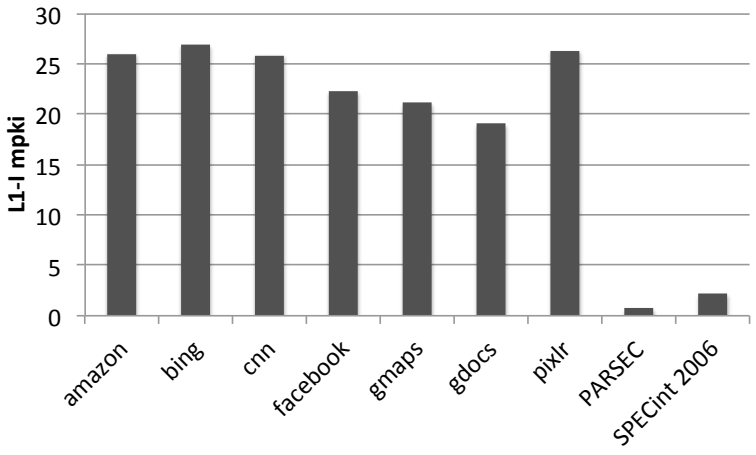


Figure 3.2: Comparison of L1-I cache mpki

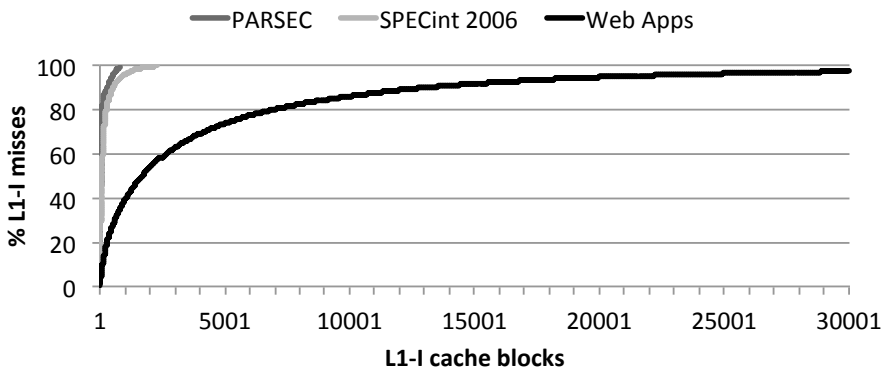


Figure 3.3: Cumulative Distribution of L1-I cache misses across cache blocks

3.3 Design

In this section I first discuss the key insights, an overview, and the architecture design of EFetch, an instruction cache prefetcher for event-driven Web applications.

3.3.1 Observations and Insights

Our design for prefetching the instruction stream ahead of its use, banks on our observation that the event signature is highly correlated with the stream of instructions executed in event-driven applications. In other words, event signature is a good predictor of the control flow inside a function, which in turn allows us to predict a function's callees and

their function bodies. In fact, we can accurately predict the order in which the callees of a function are invoked. By keeping track of the cache block addresses accessed with the event signature those cache blocks can be prefetched when the same event signature is seen again.

But, a prefetcher is good only if it can prefetch the cache blocks ahead of their use. For this reason, in our design we also keep track of the functions called (callees) corresponding to an event signature. This way, we can prefetch the next function to be called, going down the program call graph (Figure 3.5) in a depth-first manner, staying ahead of the program execution.

However, we should not keep prefetching deep down the call graph without the program advancing, because of the following reasons:

- The accuracy of predicting the next function to be executed reduces as we go further ahead of the program execution. This could lead to erroneous prefetches, polluting the cache.
- It is likely that by the time the function whose cache blocks were prefetched very early, gets called, those cache blocks might have gotten evicted. This not only results in poorer coverage, but also wastes energy.

3.3.2 Prefetching Policy

In EFetch, we use the event signature as a key, formed by a simple XOR of the event-ID with *context depth* most recent function call addresses (*context depth* = 3 in our final design). Our design depends on the predictability of the functions called and the function bodies executed under this key. So, we keep track of the functions invoked (callees) under

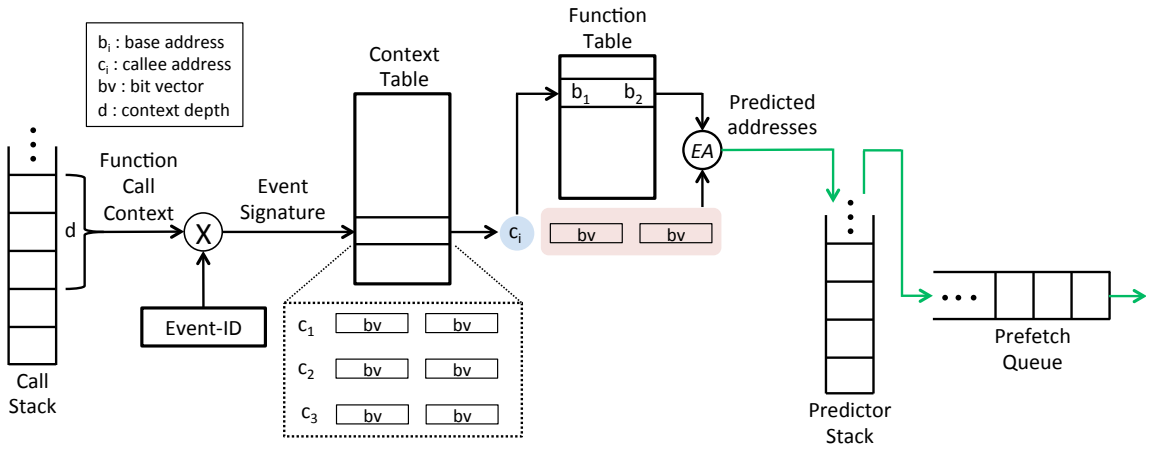


Figure 3.4: Prefetcher Design Overview.

different keys. Additionally, the function body addresses accessed within the callees are also recorded with these keys.

During program execution, on a function call, an event signature is formed from the event-ID and the call context. If this event signature has been seen before, the first callee recorded earlier is predicted to be the next function called and its predicted function body is prefetched. For example, in Figure 3.5, when f_1 gets called, g_1 is prefetched (step 0 in Figure 3.6). Since it is important for the prefetcher to stay ahead of program execution, the next predicted function is prefetched. In trying to hide memory latency, the prefetcher can, potentially, go further down the call graph and prefetch more functions ahead of time. However, we lose accuracy in predicting the next function the deeper we go down a call graph. Also, in our workloads, due to high hit rate of instructions in the L2 cache, we only need to hide the L1-I cache miss latency. Thus, EFetch prefetches only the next predicted function (except in the case of leaf functions).

When the program catches up (step 1), the prefetcher goes down the callgraph, prefetching the next function (h_1). On returning from a callee, the next callee is prefetched.

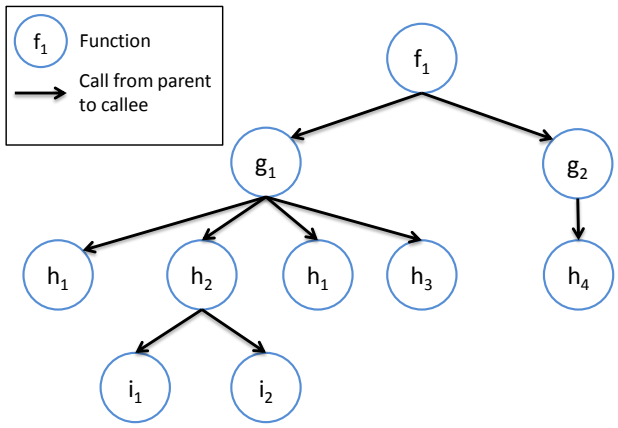


Figure 3.5: A pre-order traversal of this call graph shows the order in which the functions are called.

	Call Stack	Functions Prefetched
0	f_1	g_1
1	$f_1\, g_1$	h_1, h_2
2	$f_1\, g_1 h_1$	-
3	$f_1\, g_1$	-
4	$f_1\, g_1 h_2$	i_1, i_2
	...	

	Call Stack	Functions Prefetched	
5	$f_1\, g_1$	h_3	← h_2 returns
6	$f_1\, g_1 h_1$	-	
7	$f_1\, g_1$	-	
8	$f_1\, g_1 h_3$	-	
9	$f_1\, g_1$	-	
10	f_1	g_2	
	...		

Figure 3.6: Functions prefetched on different points of program execution

If the prefetched callee is a leaf function, we can predict the next callee (its sibling) with high accuracy. So, in case of a leaf function, EFetch prefetches its next sibling (h_2 in step 1). If the sibling is also a leaf function, that gets prefetched too and so on. The prefetcher stops if the sibling is a non-leaf function, even if there are more siblings in the callee list which are leaf functions. This is because, it is unclear how deep the call graph might go when executing the non-leaf function, and prefetching upcoming leaf siblings might be too early.

If a callee gets mispredicted (on a non-leaf function call, the function called does not

match the last prefetched function), the prefetcher stops and synchronizes itself with the actual function call stack. Thereafter, prefetches continue just like before.

3.3.3 Predicting Callees and Prefetching Future Accesses

EFetch design architecture has been illustrated in Figure 3.4. The event-ID is stored in the event-ID register. It is formed by the browser using the type of event and the JavaScript function address. *EA* is a unit that forms addresses from their compact stored representation consisting of a base address and a bit vector.

The current function call stack of the program is maintained in the *Call Stack*, each entry of which is a function call address. On a function call, the function call address is pushed onto the *Call Stack* and the event signature is computed. The event signature and the function call address is then used to index in to the *Callee Predictor* to obtain the list of callees and their function bodies to be prefetched.

The *Callee Predictor* records the function bodies and callees corresponding to different event signatures. To save space, this information is maintained in two tables (Figure 3.4), one indexed by the *context key* - *Context Table*, the other indexed by the function call address - *Function Table*. The list of callees is stored in the *Context Table*, and their function bodies to be prefetched is obtained from the *Function Table* (Section 3.3.4).

If an entry is found in the Callee Predictor, the list of callees is read out, and pushed on to the *Predictor Stack* such that the first predicted callee is on the top of the stack. The *Predictor Stack* is used to maintain the state of the prefetcher and synchronize it with the *Call Stack*. Each entry of this structure stores a function address and four status bits (Section 3.3.4).

The function body addresses of the function at the top of the *Predictor Stack* (if it has not been prefetched yet) are pushed in to the *Prefetch Queue*. Addresses present in this queue are then prefetched.

If the prediction of callees was correct, it is guaranteed that the top of the *Predictor Stack* matches the next function the program invokes. Thus, when the program invokes the next function, the function at the top of the *Predictor Stack* is validated, and prefetching under the new event signature is initiated. When the program returns from a function, the entry at the top of the *Predictor Stack* is popped.

If the prefetcher mispredicts the next function to be executed, detected by a mismatch in the function called and function present at the top of the *IPStack*, prefetching is halted. The *Predictor Stack* is cleared from the top till the parent of the current function. This is because, except till the parent of the current function, the prefetcher does not know how the call graph looks. This way, on a misprediction, the prefetcher aligns itself with the real *Call Stack*.

3.3.4 Design of Hardware Structures

Event-ID register : This register stores the event-ID uniquely identifying an event. This ID is formed by the browser using these two pieces of information - the type of event (mouse click, mouse scroll, timed event, etc.) and the JavaScript function called to handle the event. Using a special instruction (added in our design) this event-ID is stored in the event-ID register.

Call stack : This structure maintains the current function call stack of the program. Each entry in this stack is a function call address. On every function call, the call address

of the function is pushed on to this stack. Conversely, on every return, the entry at the top of the stack is popped out. It has 32 entries. If the structure overflows, the oldest entries are discarded.

Prefetch Queue : Prefetch requests are made from this structure. It maintains a queue of L1-I cache block addresses to be prefetched. We observed that the L1-I cache blocks accessed in a function are contiguous with a few discontinuities. Therefore, pairs of a base address with an associated bit vector representing the nearby blocks are recorded, similar to the scheme used in [25].

Callee Predictor : This structure is used to record the function bodies and callees with respect to event signatures. This is a combination of two tables (Figure 3.4), one indexed by the event signature - *Context Table*, the other indexed by the function address - *Function Table*. Each entry in the *Function Table* is a list of two base addresses (offsets from the function address at cache line granularity), pointing at different points in the function body. Each entry in the *Context Table* is an ordered list of three unique callees (ordered by their first call) and two bit vectors for each callee. These are callees of the function at the top of the call stack for this event signature (Figure 3.4). The callees and the bit vectors are read from the *Context Table*. The callee address is used to index in to the *Function Table*. The base addresses read out from the *Function Table* combined with the bit vectors form the function body addresses to be prefetched. Both tables have 4k entries each.

Predictor Stack : It maintains the state of the prefetcher and is used to synchronize it with the Call Stack. This is a 32-entry stack, each entry of which stores a function call address and four status bits. Following is a description of what each bit implies if it is set :

- Prefetch bit : Instruction cache blocks of this function have been pushed on to the

prefetch queue.

- Leaf bit : This is a leaf function.
- Parent Executing bit : This is a function whose prefetch requests have been pushed on to the Prefetch Queue; it has completed execution, but its parent has not.
- Mispredict bit : There has been a misprediction and this is the last correctly predicted function.

3.3.5 Prefetching Algorithm and Example

EFetch predicts the instructions to be fetched by predicting the next function to be called. Here, the **top of the stack** means the first entry in the *Predictor Stack* that does not have its Parent Executing bit set.

On a function call (Figure 3.7a), given the current function call context and the event-ID, the event signature is formed. If the top of the *Predictor Stack* matches the function called, it implies the prediction was correct. The event signature is used to index in to the Callee Predictor. If there is no entry, nothing is done. If there is an entry, the prefetcher reads the list of callees and pushes them in reverse order in to the *Predictor Stack*. This is so that the callees can be popped off the stack in the order in which they are predicted to be called.

If there is a mismatch, we try to match the function called with any entry in the *Predictor Stack* going up from the top. If it matches any entry, it means this function has been called again in the same invocation of its parent, and that we have prefetched it before. So nothing more is to be done.

If the current function does not match any entry, it means the prefetcher mispredicted

the current function call. In this case, prefetching is halted and the *Predictor Stack* is cleared until the parent of the current function. This is because, except till the parent of the current function, the prefetcher does not know how the call graph looks. Prefetching restarts once either the parent returns, or if by using the current function as part of the event signature, the prefetcher can make predictions.

On all occasions, if the top of the *Predictor Stack* has not been prefetched, prefetch requests for its function body addresses are made. Its Prefetch bit is then set marking that it has been prefetched. The prefetcher now waits for the program to catch up. We do not want to prefetch too far ahead of the program execution, unless it is a leaf function, in which case its sibling is prefetched as well.

Similarly, on a function return (Figure 3.7b), if the top of the *Predictor Stack* matches the function returned, its Parent Executing bit is set and its callees, if any, are popped. If there is a mismatch however, we try to match the function returned with any entry in the *Predictor Stack* going up from the top. If we find a match, it means the function has completed a previous call, and nothing more is to be done.

If the Mispredict bit of the top of the *Predictor Stack* is set, it implies that a callee was mispredicted and the *Predictor Stack* had already been cleared up till this entry before. Otherwise, if the top of the *Predictor Stack* has not been prefetched, follow the last step of the algorithm as in the case of a function call.

In our design, a prefetch request is made only if the cache block is not present in the L1-I cache.

Figure 3.6 shows which function is prefetched relative to program execution. Following the algorithms described in Figures 3.7a and 3.7b, Figure 3.8a shows a step-by-step walk-

// On a function (*f*) call

1. **if** top (*Predictor Stack*) == *f*
   a. Callees of *f*, if any, are pushed on to the *Predictor Stack* in reverse order
2. **else if** *f* matches any entry in the *Predictor Stack* going up from top (*Predictor Stack*)
   // *f* was called and prefetched before.
   // Nothing more to be done
3. **else**
   // the function call was mispredicted
   a. The *Predictor Stack* is cleared until the parent of *f*
   b. The *Mispredict* bit of top (*Predictor Stack*) is set
   c. *f* is pushed onto the *Predictor Stack*
   d. The *Prefetch* bit of *f* is set
4. **if** f_x (= top (*Predictor Stack*)) has not been prefetched (its *Prefetch* bit is not set)
   a. Prefetch f_x
   b. Set its Prefetch bit
   c. if f_x is a leaf function (its *Leaf* bit is set)
      i. f_x = sibling of f_x not been prefetched, go to step 4a

(a) Algorithm followed on a function call

// On a function (*f*) return

1. **if** top (*Predictor Stack*) == *f*
   a. Its *Parent Executing* bit is set
   // *f* has returned but its parent has not. Its parent is the
   // new top (*Predictor Stack*)
   b. Its callees, if any, are popped from the *Predictor Stack*
2. **else if** *f* matches any entry in the *Predictor Stack* going up from top (*Predictor Stack*)
   // *f* has completed a previous call
   // Nothing more to be done
3. **if** *Mispredict* bit of top (*Predictor Stack*) is set
   // a callee was mispredicted
   // Nothing more is done
4. **if** f_x (= top (*Predictor Stack*)) has not been prefetched (its *Prefetch* bit is not set)
   a. Prefetch f_x
   b. Set its Prefetch bit
   c. if f_x is a leaf function (its *Leaf* bit is set)
      i. f_x = sibling of f_x not been prefetched, go to step 4a

(b) Algorithm followed on a function return

Figure 3.7: Prefetching Design Algorithms

through of the states of the Predictor Stack as an example program executes. At the start of the call graph (Figure 3.5), the Predictor Stack has f_1 at the top of the stack and it has already been prefetched (Prefetch bit set).

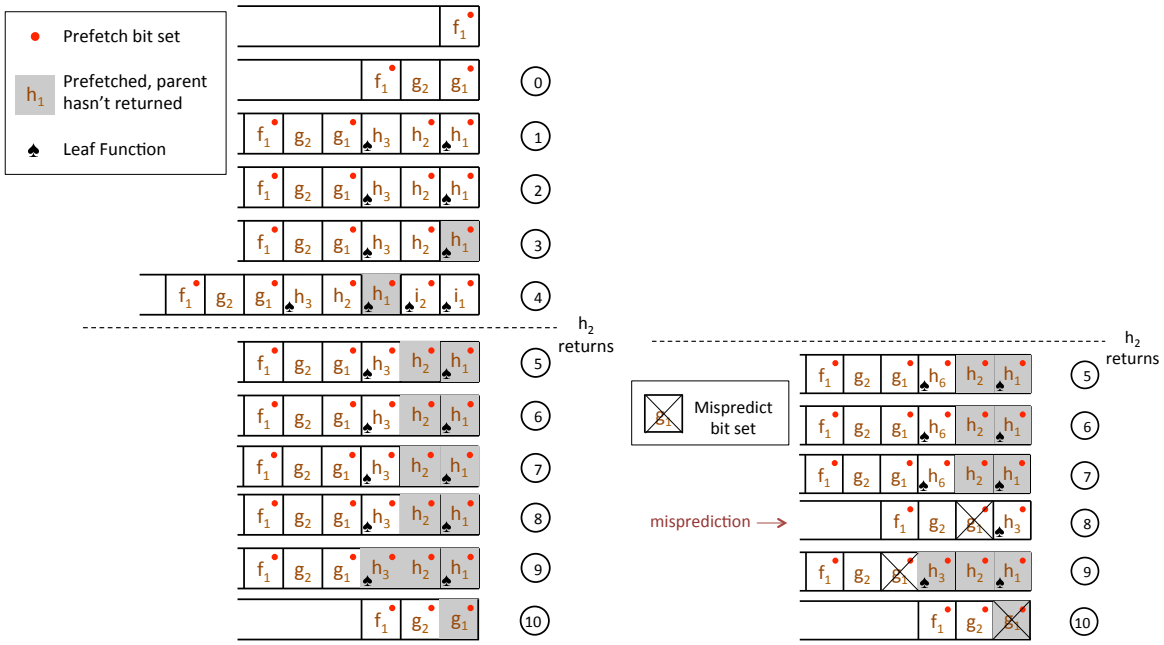
In Step 1, when g_1 is called, it matches the top of the stack, implying the prediction was correct. Its callees are pushed on to the stack. h_1 is prefetched. Since it's a leaf function, the next callee, h_2 is also prefetched. This is not a leaf function, so prefetching stops, waiting for the program to catch up. Similarly, when h_2 is called in step 4, both of its callees being leaves are prefetched, without waiting for the call to the first callee i_1.

When h_1 is called again in step 6, it does not match the top of the stack. However, it does match an earlier entry since it was its second call by its parent - g_1. This is, therefore, not a misprediction. This function has already been prefetched.

The above is a case where the prefetcher always predicted correctly. Figure 3.8b shows a case where everything proceeds as before until h_3 is called, at which point a misprediction is detected, since the next predicted callee is h_6.

When h_3 is called in step 8 the top of the stack does not match the function called or any entries with their Parent Executing bit set. A misprediction is detected - the Predictor Stack is cleared until g_1; it's Mispredict bit is set, and the newly called function (h_3) is pushed on to the Predictor Stack, with it's Prefetch bit set. This way the *Predictor Stack* is synchronized with the *Call Stack*.

When h_3 returns, the top of the stack (g_1) has a Mispredict bit set, so nothing more is done.



(a) Predictor Stack (b) Predictor Stack on a misprediction

Figure 3.8: Prefetching example

3.4 Methodology

In this work we have attempted to evaluate the behavior of event-driven JavaScript (JS) web workloads. This nature of JavaScript execution is not captured by existing JavaScript benchmark suites like Octane [2] and Sunspider [4]. Real-world JavaScript web workloads, as executed in the rendering of popular web pages, bear little resemblance to the benchmark suites, as shown in [43]. An important difference is the lack of event-driven execution in existing benchmark suites. For the above reasons, for this work, we have created workloads from real web sites, studied their characteristics and evaluated our design using them.

3.4.1 Web Applications

Table 3.1 lists the 7 real web applications that we have used in this study. These web-sites were chosen since they are both important and popularly visited, and they cut across a

Website	Actions Performed	# events executed	# inst. (millions)
amazon (amazon.com)	Search for a pair of headphones, click on one result, go to a related item	7,784	434
bing (bing.com)	Search for the term "Roger Federer", go to new results	4,858	259
cnn (cnn.com)	Click on the headline, go to world news	13,409	1,230
fb (facebook.com)	Visit own homepage, go to communities, go to pictures	9,305	2,165
gmaps (maps.google.com)	Search for two addresses, get driving, public transit and biking directions	7,298	2,722
gdocs (docs.google.com)	Open a spreadsheet, insert data, add five values	1,714	809
pixlr (pixlr.com)	Add various filters to an image uploaded from local storage	465	26

Table 3.1: Web Sites visited and actions taken, and a measure of the size of the benchmarks

diverse range of tasks users typically perform on the web browser. Our applications cover e-commerce (amazon.com), search (bing.com), news (cnn.com), social networking (facebook.com), mapping (maps.google.com), online document editing (docs.google.com) and online image editing (pixlr.com).

The actions taken on visiting each site (a browsing session), are meant to represent a typical behavior of a user on a short, but complete visit to the site. we kept the browsing

Core	4-wide, 1.66 GHz OoO, 96-entry ROB, 16-entry LSQ
L1-(I,D) Cache	32 KB, 2-way, 64 B lines, 2 cycle hit latency, LRU
L2 Cache	2 MB, 16-way, 64 B lines, 21 cycle hit latency, LRU
Main Memory	4 BG DRAM, 101 cycle access latency, 12.8 GB/s bandwidth
Branch Predictor	Pentium M branch predictor, 15 cycle mispredict penalty 2k-entry Global Predictor, 256-entry iBTB, 2k-entry BTB, 256-entry Loop Branch Predictor, 4k-entry Local Predictor
Interconnect	Bus
Energy modeling	V_{dd} = 1.2 V, 45 nm

Table 3.2: Details of the architecture simulated

sessions short, partly because it becomes logistically difficult to capture and study long-term uses of web applications, but also because some of these websites would typically be used in this manner. For example, searching for specific information or reading news headlines.

3.4.2 Workload Setup

We instrumented the open source web browser Chromium, running on Ubuntu 12.04. It uses the V8 JavaScript engine, also used in Google Chrome. Our instrumentation setup works as follows:

- we first, instrumented the C++ code used to implement the V8 JavaScript engine in Chromium. This helped me separate out the JavaScript part from the rest of the browser in the execution stream.

- Next we visit a website and perform the actions described in Table 3.1. In order to create a workload that is repeatable, we captured the instruction trace of the browser [1], using the trace-recording component of SniperSim [11]. This generates binary trace files containing information about the instructions executed, the direction of branches and the memory addresses accessed.

- Finally, we fed these instruction traces, into the SniperSim simulator, and evaluated our design.

The architecture simulated is based on a mobile system - Exynos 5250. The architectural configuration is detailed in Table 3.2.

[1] specifically, the Renderer component of the web browser

In our experiments we have simulated the code executed by the web browser for JavaScript execution. This also includes native code executed as part of library calls.

3.5 Results

In this section I first validate the premises our design is based on, going on to evaluate it, comparing it with previously proposed instruction prefetcher designs.

3.5.1 Prefetch Accuracy and Coverage

The basic premises that our design is based on are that event signatures are highly correlated with the control flow inside functions and that they are repetitive.

The first premise implies that both the body of the function and its callees are also highly correlated with the event signature. The second premise ensures that event signatures can be used to predict the callees and the function bodies. EFetch, therefore, keeps track of the list of callees of a function and their function bodies associated with the event signature.

We use the following two metrics to validate the design choices made in our prefetching scheme.

- Prefetch **accuracy** is defined as the percentage of prefetch hits (blocks that were prefetched and later were a hit in the cache) over all prefetch requests issued.
- **Coverage** is the percentage of misses that became prefetch hits as a results of prefetching - (# prefetch hits)*100/(# prefetch hits + # misses).

Each data point in all plots discussed in this section shows an average over all benchmarks.

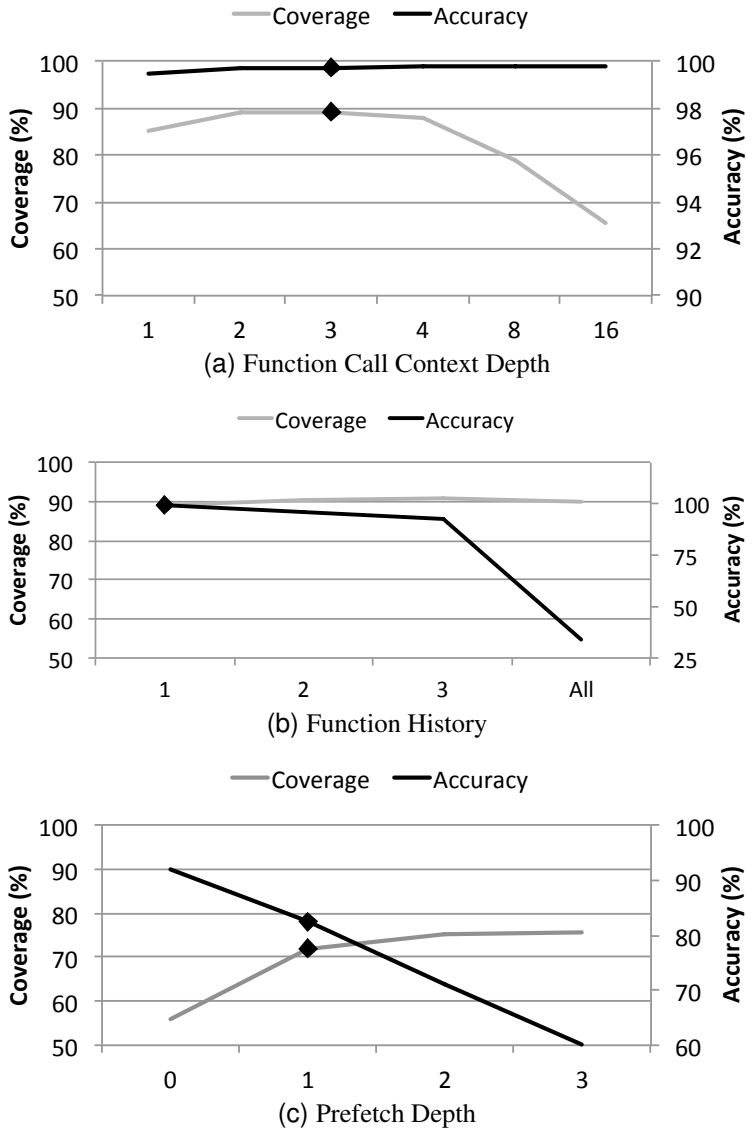


Figure 3.9: Variation in *Coverage* and *Accuracy* with varying *Function Call Context Depth*, *Function History* and *Prefetch Depth*. The diamond marks the final design choice

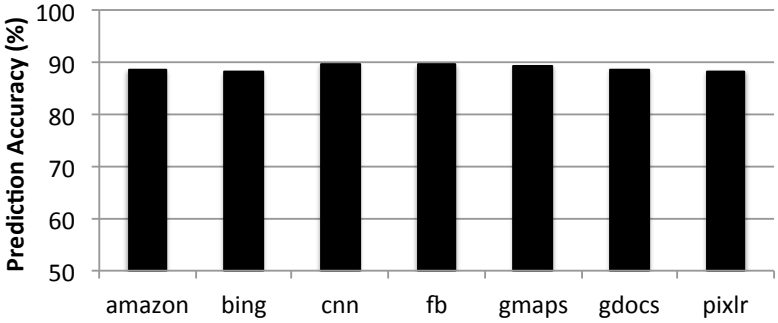


Figure 3.10: Callee Set Prediction Accuracy

3.5.1.1 Callee Set Prediction Accuracy

Any advantage to be gained from our design banks on the predictability of the list of callees using the event signature, since having made this prediction correctly, the prefetcher can prefetch the bodies of the functions predicted.

The list of callees is ordered by the first call to that callee. Maintaining an ordered list is important, since we am trying to predict the very next function to be called, staying one function ahead of program execution.

In Figure 3.10, we evaluate the predictability of the most recently seen callee set for that event signature. Here, a prediction is called accurate, if both the callees and their order is predicted correctly. As is evident from the figure, predicting that the last seen callee set for that event signature is going to be the next callee set is highly accurate.

3.5.1.2 Predicting Function Body

Having predicted the next function (callee) to be executed, it is important for the prefetcher to be able to predict the correct part of the function body to prefetch. Due to varying control flow within a function, different parts of the function body may get executed on different calls.

Given that event signature is a good predictor for callees, it is expected that it will be a good predictor for function bodies. However, this prediction is not perfect, consequently different parts of the function body might execute with the same event signature. This opens up another dimension to this problem - how much of the executed function body history to keep track of with the event signature. In Figure 3.9b, we explore precisely this point.

Here, a function history of *n*, means that we keep track of and prefetch the function body addresses executed the last *n* times this function was called with the same event signature. This figure shows a study of an ideal case, where a predicted I-cache block is brought in to the cache only when it is needed. There is no notion of timeliness - a block is considered prefetched as soon as it is predicted. This figure shows that the function body seen the last time is a good predictor of the function body going to be executed next. Using more history, does not significantly help cover more misses (coverage does not improve), but loses accuracy since it is prefetching more cache blocks that are not accessed.

3.5.1.3 Function Call Context Depth

Using function call context (and event-ID) to predict the callees of a function and their function bodies being the cornerstone of our design, how much of the call stack (number of functions) to use to form the event signature is a vital question. It is logical to believe that using more functions to form the event signature will result in better accuracy in predicting callees and function bodies, since we have more precise context information. However, there can be two problems with using a deeper context history

- It might take long for the prefetcher to learn the different contexts. Whenever we do not have the context in our table, we can not prefetch anything, thereby losing coverage.
- Using a deeper context history, results in the prefetcher using potentially stale information for a shallower context.

In our studies we found out that the first potential problem does not have a significant effect on our scheme. The prefetcher sees and is thus able to learn the frequently used

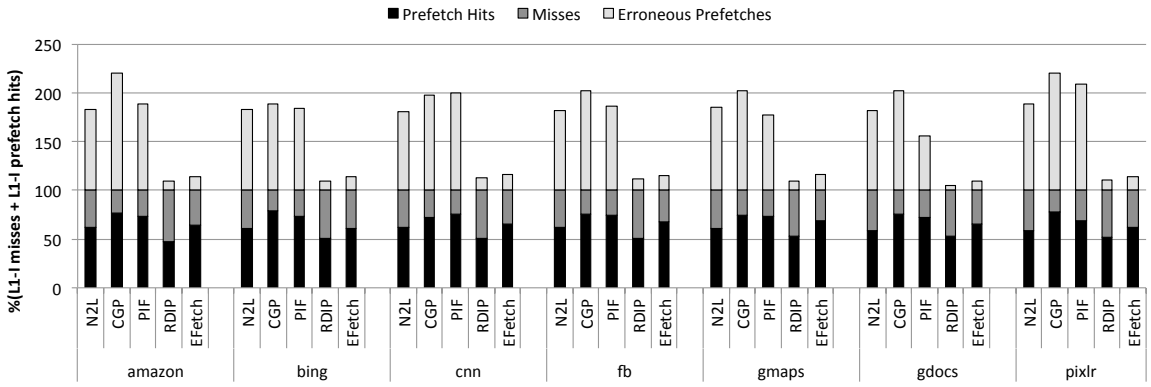


Figure 3.11: Each bar is divided in to three segments: *Prefetch Hits* are misses removed by prefetching. *Misses* are the remaining L1-I cache misses. *Erroneous Prefetches* are prefetches made which were never used (until their eviction from L1-I cache).

long contexts fairly early. The second potential problem, does have significant effects. A shallower context can appear in multiple deeper contexts. Therefore, if we use a deep context, we are likely to see older history (stale information) for the shallower context. As we have seen from the discussion in Section 3.5.1.2, using recent history results in better predictions. Using too deep a context is also counter-intuitive, since in that case we would be trying to correlate functions that may not necessarily have any bearing on the control flow of a child function. For e.g. library function calls like printf(). Its usage is common across a wide range of user functions, and it is internal call stack might have little correlation to the main() function.

Figure 3.9a validates the above hypotheses. Coverage reduces with increasing context depth without any noticeable change in accuracy.

3.5.1.4 Predicting Ahead of Program Execution

As discussed earlier, it is important for the prefetcher to stay ahead of program execution.

However, there are two issues with trying to prefetch too early - first, the cache block might get evicted by the time the program needs it; second, prediction accuracy of the prefetcher drops the further it goes ahead of the program, thereby issuing erroneous prefetches, causing pollution in the cache. Figure 3.9c confirms precisely the above arguments.

In our workloads, the L2 miss rate for instructions is 2.8%, thus for most L1-I misses we only need to prefetch early enough to hide L2 access latency. A prefetch depth of 1 is thus sufficient.

3.5.1.5 Final Design Choice

In our final design we use a context depth of 3 and a prefetch depth of 1. We only keep track of the last seen callee set and function body. A diamond marks our final design parameters in the earlier plots.

3.5.2 Performance

In this section we will compare and contrast our design with other relevant designs.

- *NP* is a design with no instruction prefetcher. This is our baseline.
- *N2L* prefetches the next 2 cache lines.
- *CGP* is our implementation of Call Graph Prefetching [8].
- *PIF* is our implementation of the Proactive Instruction Fetch [25].
- *RDIP* is our implementation of Return-Address-Stack directed instruction prefetching [31].

- *EFetch* is our design.

In Figure 3.11, we show the number of prefetch hits, misses and erroneous prefetches as a percentage of the sum of

prefetch hits and load misses. EFetch issues far fewer erroneous prefetches than PIF. Erroneous prefetches are those prefetches that never get hit before they are evicted. They hurt performance by polluting the cache kicking out potentially useful blocks, wasting L1-L2 bandwidth and consequently wasting energy. Unlike EFetch, PIF, which maintains a temporal stream of cache block accesses, has no way of knowing when the event signature has changed, and thus, keeps prefetching off the end of the temporal stream. This scheme ensures high coverage, but also suffers from low accuracy. EFetch also issues some erroneous prefetches since it assumes the last seen callee set and their function bodies will be the same as this time for a particular event signature, which is not always true. However, it recovers quickly - on the very next function call or return. N2L does not perform well. This is to be expected due to the small function sizes, large number of function calls and complex control flow inside functions.

EFetch performs better than RDIP, by achieving better coverage, even though it loses out slightly in accuracy. RDIP achieves better accuracy since its signatures are able to point at discrete call sites within functions. However, for web workloads, this leads to a very large number of signatures that need to be kept track of, overflowing their table size, causing loss in coverage. Also, RDIP only keeps track of L1-I cache misses. The exact misses can change between dynamic instances of signatures, due to cache capacity and conflict misses. EFetch keeps track of all L1-I cache accesses. For web workloads, it is

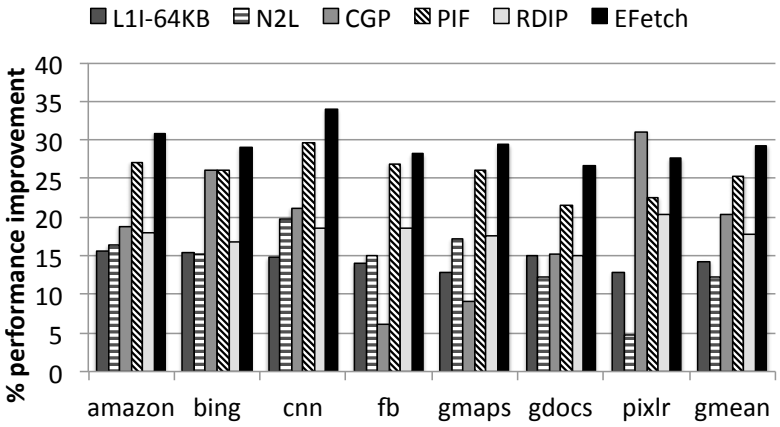


Figure 3.12: Performance achieved compared to No Prefetching (NP) as the baseline

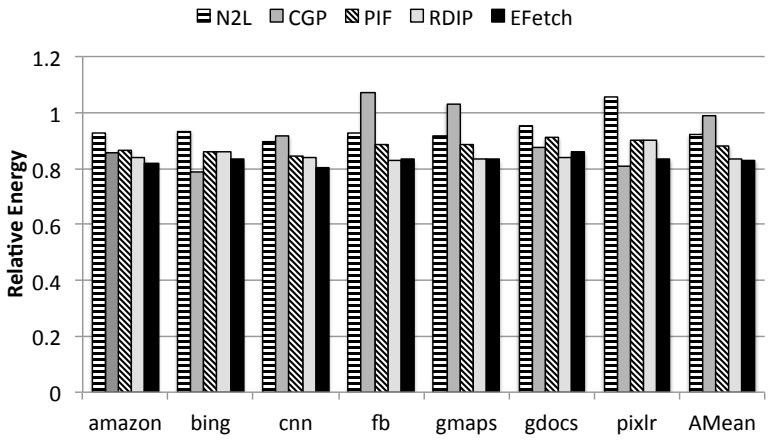


Figure 3.13: Energy expended compared to No Prefetching (NP) as the baseline

still able to keep hardware costs down due to small function sizes and better compaction of cache addresses.

Figure 3.12 shows the performance improvement of the different designs. EFetch performs as well or better than the other designs on all benchmarks. L1I-64KB has an L1 instruction cache of 64KB, roughly equal to the sum of the L1-I cache size in our baseline (NP) and the hardware storage overhead of EFetch. On average, EFetch outperforms L1I-64KB by 15%, N2L by 17%, CGP by 8.9%, PIF by 3.9% and RDIP by 11.5%.

Design	Structure	Size (KB)	Access Energy (pJ)	Static Power (mW)
CGP [8]	Call Graph History Cache	32	44.8	28.6
PIF [25]	History Buffer	136	39.3	119.9
	Index Table	68	25	62.2
RDIP [31]	Miss Table	63	32.6	49
EFetch	Callee Predictor Table	39	21.5	33.9

Table 3.3: Energy and power estimates used for hardware structures

3.5.3 Storage Overheads

We use 4K entry tables - context table, function table. Each entry in the context table stores 3 callee addresses and 2 bit vectors, along with a tag. Callee addresses are stored in the form of offsets in to the function table, using 12 bits each. The tag is 14 bits long. Therefore, a context table entry is 56 bits long.

Each entry in the function table stores 2 base addresses in the form of offsets from the function address, plus a tag. An offset needs 4 bits, tag is 14 bits in size, so each entry is 22 bits.

Both tables have 4k entries each, so a total storage of 4k*(56 + 22) bits = 39 kB is required. Comparing this with PIF, which needs 136 kB for the history buffer and 68 kB for the index table for a storage overhead of 204 kB, we get similar or better performance for 5.2 times less hardware storage cost.

3.5.4 Energy

We evaluated the energy expended by the different instruction prefetcher designs using McPat 0.8 [32]. CACTI 5.3 [54] was used to determine the per-access energy and static power of the additional hardware structures as shown in Table 3.3.[2] Figure 3.13 shows that EFetch uses less energy than all other designs, owing to its better trade-off between prefetcher accuracy and coverage. we noticed from our evaluation that the energy consumed by hardware structures added for instruction prefetching is very minimal, ranging from 0.01% of the total energy consumed for EFetch to 1.06% for PIF. Thus, to minimize energy consumed, it is not the additional hardware structures that need to be optimized, but the erroneous prefetches.

3.6 Related Work

Instruction fetch stalls cause a significant performance degradation, leading to a rich-body of work trying to solve this problem. The earliest solutions to this problem included next-line prefetching, taking advantage of sequential instruction fetch [6]. This initial idea was expanded upon and next-N-line and instruction stream prefetchers were proposed [30, 42, 57, 48] using varying kinds of events to control the aggressiveness and lookahead of the prefetcher. Next–line prefetchers are simple and work well for sequential code. However, they have poor accuracy and are not able to prefetch ahead in time for code with frequent branches and function calls.

To be able to more effectively prefetch instructions ahead of time in branch and call

[2] I have validated these numbers with the authors of RDIP

heavy code, several branch predictor based prefetchers [15, 44, 45, 51] have been proposed. Run-ahead execution [38], wrong-path instruction prefetching [41] and using an idle thread [33] or speculative threads [53, 61] can be used to generate future instruction accesses. Ferdman et. al. [25] showed that by using the instruction fetch sequence instead of the commit sequence, these approaches suffer from interference caused by wrong-path execution. Also these don't have sufficient lookahead when the branch predictor traverses loops.

The discontinuity prefetcher [50] handles fetch discontinuities. It's able to alleviate some of the issues with other branch-predictor based prefetchers by operating at an instruction block granularity. But it's lookahead is limited to one fetch discontinuity to avoid over-prediction. The branch history guided prefetcher (BHGP) [51] keeps track of branch instructions and imminent instruction cache misses, which are prefetched upon the next occurrence of the branch. However, they cannot differentiate between different invocations of the same branch (which will have a bearing on the branch outcome and instruction cache misses seen) leading to lower coverage or accuracy. Our work, differs from the above in that it targets event-driven web applications, where each event can be completely independent of the others, thereby executing completely different code from one event to the next. Also, we use event-ID and the function call stack to predict the callees of function and their function bodies, and thus is neither affected by wrong-path execution nor is unable to differentiate between different invocations of the same function.

PIF [25] addresses the limitations of branch-predictor directed prefetching by recording temporal instruction committed streams and fetch misses. By using the committed stream of instructions, it remains unaffected by the predictability of individual branches

and wrong-path instructions. To similarly be able to avoid disruptions caused by wrong-path instructions, our scheme updates the state of the prefetch structures as call and return instructions commit. Instead of using temporal streams of instructions, I utilize the event-ID and call graph information to predict the instruction cache blocks.

PIF needs to keep track of a large window of instructions committed to be able to predict the temporal stream accurately. The size of the hardware structures exceeds 200 kB, while our design needs less than 40 kB.

A recently proposed instruction prefetcher, RDIP [31], exploits the information stored in the return address stack (RAS) and uses the return addresses of functions called, to predict and thereby prefetch the next segment of function executed. This scheme was evaluated on traditional server workloads. RDIP relies on function call context being a good predictor of the next function called. Unlike RDIP, EFetch targets event-driven web applications. To do so, it utilizes the event-ID along with the function call context (event signature) to predict the next function executed. In addition to differences discussed in Section 3.5.2, EFetch prefetches only the part of the function body executed the last time with the same event signature, avoiding erroneous prefetches. RDIP, on the other hand, accumulates L1-I cache misses incurred over all previous instances when the signature was seen.

Annavaram, Patel and Davidson used the current function to guide instruction prefetching for database applications in [8]. EFetch, on the other hand is designed for event-driven web applications. By making use of the event-ID and multiple functions in the call context, EFetch is able to differentiate between invocations to the same function, and there is merit in this scheme, since it is a good predictor of varying control flow inside the function.

Also, EFetch fetches only that part of the function body, that it predicts will get executed. This is of significant importance for event-driven web workloads which are composed of a large number of very small functions, with some frequently executed very large functions. Importantly, on a single invocation of these large functions, only a small part of them is executed and it is composed of discontiguous I-cache blocks. An approach like the one in [8], would hurt performance by, on the one hand, fetching I-cache blocks beyond the boundary of small functions, and on the other hand, fetching contiguous cache blocks of the few very large functions, even though many of these I-cache blocks will go unused.

3.7 Conclusion

Event-driven web applications are becoming a dominant set of programs used in client-side computing. Unfortunately, processor architecture optimizations studied in the past are not designed to take advantage of the unique characteristics of event-driven web applications.

In this chapter we identified L1 instruction cache misses to be an important performance bottleneck in the web applications. This issue is significantly more severe than it is for conventional applications. We proposed a new prefetcher that used event signatures (formed from function calling context and event-ID) to accurately prefetch instruction cache blocks. We demonstrated that the proposed prefetcher outperforms past designs, and also has modest storage requirements.

CHAPTER IV

Accelerating asynchronous programs through Event Sneak Peek (ESP)

The asynchronous programming model presents unique opportunities that can be exploited to improve performance, beyond instruction fetch.

This chapter proposes the Event Sneak Peek (ESP) architecture to mitigate micro-architectural bottlenecks in asynchronous programs. ESP exploits the fact that events are posted to an event queue before they get executed. By exposing this event queue to the processor, ESP gains knowledge of the future events. Instead of stalling on long latency cache misses, ESP jumps ahead to pre-execute future events and gathers useful information that later help initiate accurate instruction and data prefetches and correct branch mispredictions. We demonstrate that ESP improves the performance of popular asynchronous Web 2.0 applications including Amazon, Google maps, and Facebook, by an average of 19%.

4.1 Introduction

Asynchronous or event-driven programming [5] has become the dominant programming model in the last few years. We consider the Web 2.0 applications as a case in study. Web applications are increasingly popular today as it allows users to easily run them within their web browsers and access data stored in the cloud. These web applications, written in a scripting language (mostly JavaScript), receive asynchronous input from diverse input sources such as user clicks, servers over the network, microphone, camera and other sensors. Web development platforms use the asynchronous or event-driven programming model as it is a natural fit for handling this rich array of asynchronous input.

In spite of the ubiquitous use of asynchronous programming, today's general-purpose processor architectures are heavily optimized for the synchronous programming model. As the processor industry continues to embrace heterogeneous processor designs with domain-specific accelerators as an answer to the end of Dennard scaling, we envision that at least a few of the processor cores could be customized for efficiently executing the asynchronous programs. To this end, this chapter proposes a novel processor optimization for accelerating asynchronous programs.

Asynchronous programs present unique challenges and opportunities for performance optimization. Unlike a synchronous program that executes one long (billions of instructions) task after another, an asynchronous program's execution consists of many short (only millions of instructions) events orchestrated by the timing of external events (e.g., user click). Such fine grained interleaving of many small and varied tasks destroys instruction and data locality, and exposes little repeatable patterns assumed by processor components

such as the branch predictor. Cold misses, which are of little concern in synchronous programs, constitute a significant source of pipeline stalls in asynchronous programs with short events. Finally, asynchronous programs, especially web applications, tend to exercise large instruction footprints to support a rich set of features, resulting in significantly higher instruction cache miss rates than synchronous programs.

Fortunately, asynchronous programs also provide us with new opportunities for optimizations that are absent in synchronous programs. In an asynchronous program, events are enqueued in an event queue before they are processed sequentially by a thread. By exposing the event queue to the processor, the processor can know the sequence of events that are going to be executed in the future. Our insight is that this knowledge can be exploited to take a "Sneak Peek" into the executions of the future events and gather information about them ahead of time. This information can then be used to significantly improve the accuracy of various hardware predictors when an event is executed.

We propose the Event Sneak Peek (ESP) architecture that exploits the above insight to significantly reduce instruction and data cache misses, and branch mispredictions for asynchronous programs. On encountering a long latency last-level cache (LLC) miss for instruction or data, ESP uses the idle CPU cycles to "jump ahead" to the next event in the event queue and speculatively pre-execute it. While pre-executing an event, ESP records the sequence of instruction and data cache blocks accessed, and also the branch mispredictions. ESP jumps back to the normal execution once the LLC miss gets resolved. Later, during the normal execution of a pre-executed event, the previously recorded information is used to initiate accurate prefetches and correct branch mispredictions.

Pre-execution of future events is speculative as they may depend on the earlier skipped

events. However, since the events perform varied tasks, they are fairly independent of each other. Thus, speculative pre-execution closely matches the normal execution and helps collect accurate predictions. ESP does not use the computed results from the speculative pre-execution of event, as it may require fairly complex hardware such as the components used in Thread-Level Speculation (TLS) [27, 35, 49, 52] to check for dependencies between events and recover from mis-speculations.

We devised several architectural solutions to realize an efficient ESP design. First, ESP can jump ahead multiple events. This is useful when an LLC miss is encountered while pre-executing an event after having already jumped earlier. Thus, as long as there are events in the queue, the processor can utilize the idle cycles due to LLC misses for pre-executing future events.

Second, an event may encounter several LLC misses, offering several opportunities to jump ahead to the next event. To avoid always jumping to the beginning of the next event, ESP provides the capability to save the execution context of a pre-executed event and return to that intermediate state on the next LLC miss.

Third, ESP employs small instruction and data cachelets (L0) for exclusive use during speculative pre-execution of events. Cache blocks accessed during pre-execution are brought directly to the cachelets, skipping L1 and L2, and they are never written back. These cachelets help ensure correctness by isolating the speculative store updates during a pre-execution from the normal event execution. They also help improve performance by not polluting the L1 and L2 caches for the normal event execution. Furthermore, they support efficient event pre-execution by retaining its working set, even when the processor control switches back and forth between normal and speculative pre-executions.

Finally, as ESP skips potentially thousands to millions of instructions when it jumps to the next event, fetching cache blocks directly into L1/L2 during an event's pre-execution would be too early, and hence wasteful. To address this problem, ESP uses hardware lists for collecting cache block addresses and branch mispredictions in a compressed format during an event pre-execution. The recorded information is later used during event's normal execution to initiate timely prefetches and correct branch mispredictions.

ESP could be considered as a form of runahead execution [22, 9, 38, 37], though there are some fundamental differences. Runahead execution pre-executes independent instructions that follow only a LLC *data* cache miss. Since runahead would stall on an instruction cache (I-cache) miss, it does not help improve the I-cache performance. In contrast, on encountering an instruction cache miss, ESP can skip the current event, and start fetching instructions from a different location for the next event. Second, runahead is limited by the number of independent instructions that it can discover after a load miss. ESP is more effective in finding independent instructions to pre-execute, as events tend to be independent of each other. Finally, while runahead pre-executes instructions that immediately follow a load miss, ESP may skip millions of instructions as it jumps ahead to the next event. This mandates several new solutions discussed earlier.

We have demonstrated the utility of the ESP architecture for accelerating the Web 2.0 applications, but it should be more broadly applicable for optimizing other event-driven systems. Our benchmark suite consists of a popular representative website for each type of web applications: e-commerce (`amazon`), interactive maps (`google maps`), search (`bing`), social networking (`facebook`), news (`cnn`), utilities (`google docs`), and data-intensive applications (`pixlr`).

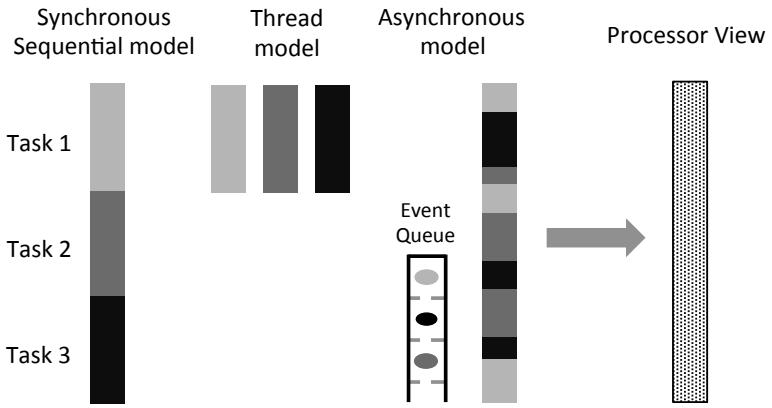


Figure 4.1: Comparison of Programming models. A processor's simplistic view of an asynchronous program's execution.

We show that ESP can improve the performance of a baseline architecture with a next-line instruction prefetcher and our implementation of Intel's DCU next-line data prefetcher [21] by 19.1%, whereas runahead improves it by only 5.7%. ESP achieves this by reducing L1 I-cache misses per kilo-instructions (MPKI) from 17.5 to 11.6, L1 data-cache miss rate from 3.2% to 1.8%, and branch misprediction rate from 12.9% to 7.1%. Since ESP executes additional instructions during pre-execution of events, it increases energy overhead by 8%.

4.2 Background and Motivation

This section distinguishes asynchronous programs from synchronous programs, and discusses why they perform poorly on conventional processor architectures. It also describes asynchronous web applications, which we use in our experiments to demonstrate the effectiveness of the ESP architecture.

4.2.1 Asynchronous Vs Synchronous Programming Models

For the purpose of illustration, let us consider a program consisting of three distinct tasks shown in Figure 4.1. A typical single-threaded synchronous program completely finishes a task before starting another. In a multi-threaded synchronous program, the tasks can be assigned to different threads. These threads may either run concurrently on different processor cores as shown in Figure 4.1, or the OS may interleave some of them at a coarse granularity on a single core. In both models, as the same task is executed for long periods of time (typically billions of instructions), they exhibit a high degree of locality and repetitive behavior, which current processors exploit to achieve high cache performance and branch prediction accuracy.

In computing systems with frequent long latency blocking operations (mostly due to I/O), synchronous programs waste processor resources waiting for those operations to complete. An asynchronous program avoids this problem by splitting a task into multiple sub-tasks at the boundaries of blocking operations and associates them with event conditions. When the required event condition is met (e.g., user click), the corresponding sub-task (event handler) is enqueued into an event queue. Instead of blocking on an I/O operation, an asynchronous program switches to execute a ready sub-task from the event queue. Programmers also take care to ensure that each event execution is short, because otherwise the system may become unresponsive.

Such a fine-grained interleaving of short (only millions of instructions) sub-tasks (events) from different tasks (Figure 4.1) destroys instruction and data cache locality in asynchronous programs. Also, it exposes little repetitive behavior, which is critical for achieving predic-

tion accuracy. Unfortunately, conventional processors are presented with a simplistic homogeneous view of an asynchronous thread (Figure 4.1), which is far removed from reality. In this paper, we propose to remedy this inconsistent view of the processor, by exposing the event queue and the event execution boundaries, and use that knowledge to improve locality and branch prediction.

4.2.2 Illustration: Asynchronous Web Application's Execution

Figure 2.1 shows an example execution of an asynchronous web application. Events are enqueued into an event-queue, from where they are dequeued by a looper thread, one at a time, and are thereafter, processed by invoking the necessary JS handler. Section 2.1 describes this mechanism in more detail.

The events get queued as soon as they are generated. However, we have observed that they remain in the event-queue for several tens of microseconds before getting dequeued and executed. This duration can be used to pre-execute the future events, which can help improve their performance when they finally execute. Also, as shown in Figure 2.1, events perform varied tasks, and as a result exhibit high degree of independence. This property allows us to speculatively pre-execute future events with high accuracy.

4.2.3 Asynchronous Web 2.0 Applications

Web 2.0 has revolutionized the Internet experience. This has been enabled in large part by the shift of computation from servers to clients. This shift, along with the increased complexity of web sites in an effort to provide a rich user experience has dramatically increased the computation responsibilities of the clients (mobile and desktop). As a result,

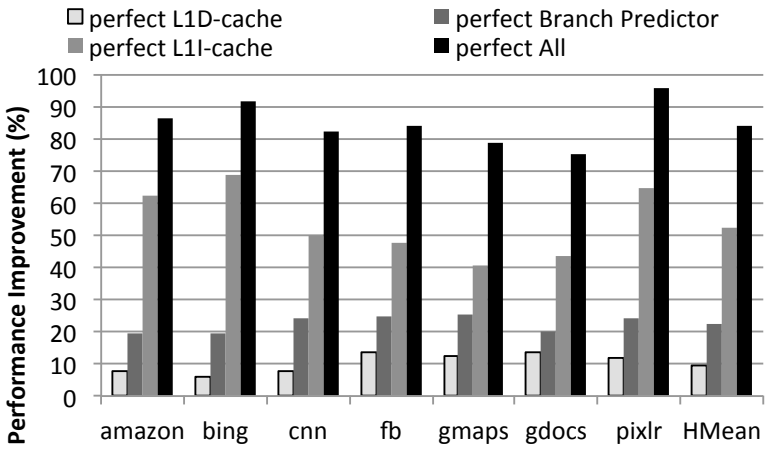


Figure 4.2: Performance potential in web applications.

the proportion of JS execution in the renderer process has almost doubled in the last ten years during the transition from Web 1.0 to 2.0, and is only likely to grow further.

Asynchronous execution of client-side JS is responsible for, among other things, handling user interaction with the web browser and providing dynamism to the web page content. It commands a lion's share of the client-side computation, and directly affects browser response time as it can block further user interaction. Even in web applications that require network transactions, JS computation accounts for more than 20% of response time [18]. Thus, improving JS performance in web browsers is not only useful for compute-intensive web applications, but can also help improve response time.

For the reasons stated in Section 4.2.1, asynchronous web application incur very high L1-I cache miss rates and branch misprediction rates, with moderate L1-D cache miss rates. By using perfect caches and branch predictors, we can nearly double the performance of asynchronous web applications (Figure 4.2).

However, existing prefetching techniques are either incapable of alleviating the problem or come at too big a hardware cost. Our goal is to expose the event-queue to hardware and

use that information to accelerate asynchronous applications.

4.3 Event Sneak Peek (ESP): Design

Conventional micro-architecture designs have ignored the unique challenges and opportunities posed by asynchronous programs. This section presents our insights for exploiting event-level parallelism in asynchronous programs for improving ILP. We then present an overview of the important components in our Event Sneak Peek (ESP) architecture, which exploits our insights to accelerate asynchronous programs.

4.3.1 Exploiting Event-Level Parallelism

By exposing the software event queue to the hardware, the processor can know the events that are going to be executed in the future. Since events are fairly independent of each other, on encountering a long latency LLC (last-level cache) miss while executing an event, instead of stalling the pipeline, the processor can "jump ahead" to speculatively pre-execute the first event waiting in the queue. If another LLC miss is triggered while speculatively pre-executing an event, the processor can jump ahead one more time to the second event in the queue. As we later show in Section 4.6.6, we did not find many opportunities to pre-execute a third pending event in the queue. Therefore, we made an important decision to support only two event jump aheads at any time. Eventually, when the LLC miss is resolved, the processor can jump back to continue the normal execution.

Pre-execution of a future event is speculative, because there is a small chance that it might be dependent on the previous events that were jumped over. Therefore, if we want to

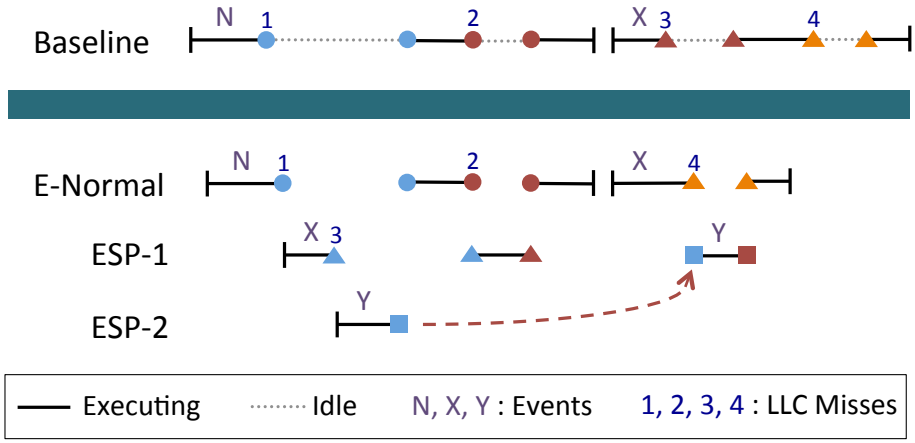


Figure 4.3: Illustration: ESP Execution

reuse the computation from the pre-execution of an event, we would have to invest in some fairly complex hardware to check for any dependencies between events and recover from mis-speculations.

Instead, we design a simpler optimization where speculative pre-execution of an event is used to only collect the instruction and data addresses it accessed, and the branch outcomes. The collected information is later used when the event is executed in the non-speculative mode to initiate accurate prefetches and correct branch predictions. In this way, we exploit event-level parallelism in asynchronous programs to improve their ILP.

4.3.2 Event Sneak Peek: Illustration

Figure 4.3 illustrates an example execution in ESP. It shows a state where the processor is executing the current non-speculative event `N`, in the normal mode (`E-Normal`). On encountering an LLC miss (`1`) for `N`, ESP skips the current event and jumps ahead to speculatively pre-execute the first waiting event (`X`) in the queue. We refer to this execution mode as `ESP-1`. If another LLC miss (`3`) is encountered while pre-executing the event `X`, or if the event `X` ends, the processor jumps ahead one more time to the `ESP-2` mode,

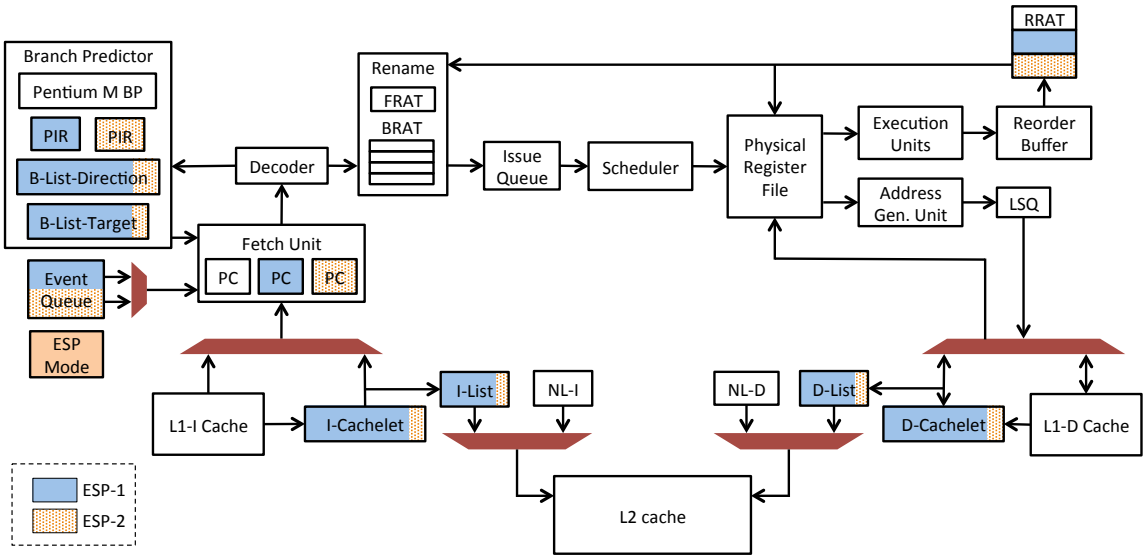


Figure 4.4: The ESP Architecture

where it pre-executes the second event in the queue (`Y`). The instruction and data addresses accessed, along with the branch outcomes, during the ESP modes are recorded. Eventually, when the LLC miss resolves for the normal event `N`, the processor goes back to continue its non-speculative execution.

If the normal event `N` encounters another LLC miss (`2`), the processor again jumps ahead to `X` and starts pre-execution from the point where it had left off during the last visit.

When the current event `N` ends, the software dequeues `X` and enqueues `Z`. Thus, the event `X` is now the current non-speculative event, and the events `Y` and `Z` correspond to the `ESP-1` and `ESP-2` modes respectively. The prediction information gathered for `X` during its pre-execution is used to improve prefetching and branch prediction in the normal mode. Thus, event `X` finishes earlier in ESP, compared to the baseline processor.

4.3.3 Design Overview

Figure 4.4 illustrates the Event Sneak Peek (ESP) architecture. ESP consists of a hardware event queue that holds the event handlers (starting instruction addresses) for the future events. Software is responsible for enqueing and dequeing event handlers into this queue through two new instructions that we add to the ISA. The `ESP-mode` status register specifies the event that is being executed, which can either be the non-speculative current event or one of the future events in the hardware event queue.

The rest of the ESP's architectural extensions can be broadly classified into three main categories. The first set of components help preserve the execution contexts of the current event and the speculatively pre-executed future events. The second set of components help collect branch outcomes and addresses accessed during the pre-execution of an event in the sneak peek mode. The final set of modifications enhance existing prefetchers and branch predictor with the information collected by pre-executing events. The rest of this section provides an overview for each of these three categories.

4.3.4 Persisting Event Execution Contexts

ESP supports speculative pre-execution of future events to be re-entrant. It is possible for the current event to encounter several LLC misses. When the execution jump ahead to the next event on an LLC miss, instead of starting from the beginning of the event, the pre-execution continues from the point where it was suspended during the last visit. This allows ESP to pre-execute deep into the future events.

To allow control to switch back and forth between the non-speculative current event

and the speculative future events, ESP maintains additional execution contexts (registers, caches, and global branch history). The number of additional execution contexts required depends on the number of events that ESP allows to jump ahead at any given time, which is limited to two as we mentioned earlier (Section 4.3.1).

Registers: The baseline register context supports the current non-speculative event. In addition, ESP supports two sets of additional register contexts, where each set includes a retirement register alias table (RRAT), and special purpose registers (e.g. program counter (PC), stack pointer (SP)).

I and D-Cachelets: A naive ESP design would utilize the caches in the baseline processor to support event pre-executions. However, this poses both correctness and performance challenges. To ensure correctness, a store pre-executed by a future event in an ESP mode should not be allowed to update the cache and memory state, because otherwise the earlier non-speculative event may see an incorrect value. Simply dropping the speculative store updates during pre-execution is also not a feasible solution, because then the loads during pre-execution may not return correct values.

To solve this problem, ESP has two small data cachelets (D-cachelets), one for each ESP mode, which act as the L0 cache used exclusively in the ESP modes. Stores in the ESP modes update the corresponding D-cachelets. However, cache blocks in D-cachelets are not written back to L1 and beyond, which ensures correct memory state for the earlier non-speculative event.

D-cachelets provide significant performance benefits. A cache block fetched to service a load or a store in an ESP mode, bypasses the caches and is brought directly into the corresponding D-cachelet. This avoids polluting the L1 cache state for the non-speculative

event and the D-cachelet state for the other ESP mode. D-cachelets also help improve performance in the ESP mode by preserving the working set for the pre-executed events as the control switches back and forth between different ESP modes.

The sizes of the two D-cachelets are provisioned to be large enough to capture about 95% of reuse in the ESP modes. Since the processor spends much less time in the ESP-2 mode (two event jump ahead) compared to the ESP-1 mode (one event jump ahead), its D-cachelet can be a lot smaller than the one used for ESP-1 mode. Through empirical analysis (Section 4.6.6), we find that 5.5 KB D-cachelet for ESP-1 mode, and 0.5 KB D-cachelet for ESP-2 mode is adequate.

Since instruction caches are read-only, we do not need instruction cachelets (I-cachelets) for correctness. However, ESP uses two I-cachelets (5.5 KB and 0.5 KB) for the two ESP modes to reap the performance benefits noted above for D-cachelets.

Branch Predictor Context: Our baseline processor models Pentium M branch predictor [56], which uses a path information register (PIR). We find that preserving the small PIR states across control switches between events can result in significantly more accurate branch predictions. Therefore, ESP uses two additional PIRs for the two ESP modes. Preserving the rest of the branch predictor state did not improve prediction accuracy significantly enough to warrant the high area cost it would entail. Thus, the rest of the branch predictor structures are updated just like in the baseline processor in both the normal and the ESP modes.

4.3.5 Prediction Lists

As explained earlier in this section, the events in the ESP modes should not use the L1 caches due to correctness and performance challenges. While data updates cannot be made visible to the rest of the system, bringing cache blocks with read-only permissions to the L2 cache is a viable option. However, bringing these cache blocks even to the L2 is premature, especially in the case of a long running current event. Moreover, this only solves part of the problem, since even when these cache blocks stay in the L2 when they are accessed by the current event, the processor still has to pay the penalty of an L2 cache access.

The cachelets (I and D) are designed to be large enough to only hold the current working set of the pre-executed events, so that the events in ESP modes can execute faster, and not for the working sets of the events to persist in them when the events execute in the normal mode.

Even if the cache blocks of speculatively pre-executing events were brought in to the L1 caches, these fetches would be too premature, and the cache blocks would get evicted by the time the event needs them while executing in the normal mode. Also, the instruction and data footprints of long events are, anyway, too large to fit in even the L1 caches. Therefore, ESP uses two sets of two hardware lists (I-List and D-List) to record the cache block addresses accessed during the two ESP modes. The records also contain the time (measured as instruction count from the beginning of the event) of those cache accesses. These are later used to determine the time for prefetch requests when the event executes in the normal mode.

To increase the performance of the branch predictor, simply training it during the ESP

modes is not sufficient. Such a scheme would train the branch predictor on branch outcomes of a sizeable initial part of the event. If this is used during the event's execution in the normal mode, for the initial branches, the branch history will include branches much further into the event, and completely irrelevant to the current branch instructions. As the event execution progresses, the branch predictor will get trained on the recently past branches on top of its initial training. It will thus get trained on a medley of branch instructions from the future and the past, degrading its prediction accuracy. However, if we are able to keep the branch predictor trained on branch outcomes of just enough future branches, it will have trained on the most relevant branches to perform well.

To this end, the instruction addresses and directions of branches are stored in a list (B-List-Direction), along with the targets of taken branches (B-List-Target). Similar to the I and D-lists, B-List-Direction records the time of the branches to more accurately guide branch prediction during the normal mode.

The lists for the ESP-1 mode are much larger than the lists for the ESP-2 mode, as its pre-execution can go much deeper.

4.3.6 ESP Predictors

When an event executes in the normal mode, information that was previously recorded in an ESP mode is used to prefetch and correct branch mispredictions.

To ensure timely prefetches, ESP issues prefetches from the I-list and D-list a preset number (190) of instructions in advance of its use (determined by the instruction count stored in the list entry), or the earliest possible. One challenge is initiating prefetching before an event starts so that cache misses during the initial execution of an event are avoid.

We solve this problem by exploiting our observation that a looper thread (one that dequeues and processes events from the event queue) executes extraneous instructions (about 70) for event queue management towards the end and also before beginning an event. We use this opportunity to start initiating prefetches at least 70 instructions before an event starts.

These prefetch requests override those from the baseline prefetcher, next-line (NL). Once the addresses in the list are exhausted, NL takes over. Therefore, for long events, ESP would initially use the lists issuing accurate prefetch requests, but later has to rely on the baseline prefetcher.

The instruction addresses and branch outcomes stored in B-List-Direction, along with the targets in B-List-Targets, are used to train the branch predictor with several branches ahead of their execution. The training is kept loosely coupled with the actual branch execution, a preset number of branches ahead, so that the branch history is neither too far in the future nor too short, so as to enable accurate branch prediction. Once the list entries are exhausted, the branch predictor works as it normally would, without the help of just in time training.

4.4 Implementation Details

This section first describes how the different architectural extensions of ESP interact with each other to realize this design, followed by the details of the added hardware components. It concludes with a description of rare cases that arise during ESP modes and how they are handled.

4.4.1 Switching Event Execution Contexts

The events to be executed subsequently are present in a software event-queue. We propose that arguments to the event handler be passed as data members of an object, whose address is passed to the event handler. With this change, the software event-queue is exposed to the hardware in a 2-entry register-like structure which keeps track of the next two events. Each entry of this *Event Queue* holds the starting address of an event handler, the argument object address and an execution-underway (EU) status bit, which indicates if speculative pre-execution of this event is already underway or not. We propose addition of function intrinsics to the system to enqueue and dequeue events from the Event Queue.

The `ESP-mode` status register specifies the event that is being executed, which can either be the non-speculative current event (normal mode) or one of the future events in the hardware event queue (ESP modes).

Entering ESP mode: The processor enters ESP-i mode when a memory operation misses in the last-level cache (LLC) and the corresponding instruction reaches the head of the Re-Order Buffer (ROB), where "i" is 1 on switching from the normal mode, and 2 on switching from the ESP-1 mode (these values of "i" have been used in the rest of the section to describe the hardware components used).

The address of the instruction that caused the LLC miss is recorded in the duplicated Program Counter register for ESP-i, PC_{ESP-i}. This instruction and all following instructions of this mode are drained from the pipeline without committing them, similar to how wrong-path instructions in the case of a branch misprediction are handled.

To correctly resume execution on return to a mode, ESP preserves the register state of

each context, by duplicating the Retirement Register Alias Table for each context, $RRAT_{ESP-i}$, while using one common physical register file. While the pipeline is being drained, before entering ESP-i mode, RRAT_{ESP-i} is copied onto the Front-end RAT (FRAT), which restores the register state of ESP-i mode.

Execution in ESP mode: If the event is starting its execution in the ESP-i mode (EU bit is not set), the address of the instruction to fetch is specified by the corresponding Event Queue (and the EU bit is set). On the other hand, if it is resuming execution, PC_{ESP-i} holds the next instruction address.

In ESP-i mode, instruction fetches use the *I-cachelet*$_{ESP-i}$, as an L0-I cache, bypassing the L1-I and L2 caches. Addresses of I-cache blocks fetched by ESP-i are stored, in order, in *I-list*$_{ESP-i}$. These are later used for instruction prefetching when the event executes in the normal mode. Data requests use their counterpart structures in exactly the same fashion.

Exiting ESP mode: When the LLC miss of the current event is resolved, the processor returns to its current execution. Similar to the manner in which a branch misprediction is handled, all instructions in the pipeline are flushed at this point, in an effort to return to the execution of the non-speculative event as quickly as possible. To restore the register context, RRAT of the current event is copied on to the FRAT. The Return Address Stack (RAS) is cleared, since it might contain return addresses of functions in ESP-i modes.

4.4.2 Enhancing Cache Performance

I and D-cachelets isolate cache blocks accessed during the ESP modes from the rest of the system for correctness and performance purposes, as explained earlier. On the other hand, I and D-lists record cache block addresses and enable timely prefetches enhancing

cache performance.

Cachelets: The sizes of the two cachelets are provisioned to be large enough to capture 95% of reuse in the ESP modes. Both I and D-cachelets are 12-way set associative caches, of size 6 KB each. Since the processor spends much less time in the ESP-2 mode (two event jump ahead) compared to the ESP-1 mode (one event jump ahead), its cachelet can be a lot smaller than the one used for ESP-1 mode. Through empirical analysis (Section 4.6.6), we find that 5.5 KB I-cachelet for ESP-1 mode, and 0.5 KB I-cachelet for ESP-2 mode is adequate (same for D-cachelets).

To isolate the two speculative events from each other, one way of the cachelets is reserved for ESP-2 (first way). When the current event finishes execution, the event in ESP-2 mode moves to ESP-1 mode, while the next event in the Event Queue will be executed in ESP-2 mode. The last way is now reserved for ESP-2. The event now in ESP-1 mode, gets ten more ways of cachelet space, plus its original reserved way. In this manner, the way reserved for ESP-2, switches between the first and the last way, on completion of events.

Lists: The I and D-lists are used to store the corresponding cache block addresses accessed during ESP modes, such that those cache blocks can then be prefetched when the event executes in the normal mode.

I-list stores the list of I-cache block addresses accessed by retiring instructions in the ESP modes. Each entry is composed of an I-cache block address stored as an offset from the previous list entry (8 bits), the number of contiguous cache blocks accessed after it (3 bits), the number of instructions executed before accessing this block, stored as an offset from the previous list entry (7 bits) and a large offset bit indicating that this cache block is much further away from the previous list entry than can be encoded in an offset of 8 bits.

In such a case the next two list entries specify the complete 26-bit cache block address.

The list is physically divided in to two circular queues, 499 bytes for ESP-1 and 68 bytes for ESP-2. When the current event finishes execution, the event executing in ESP-2 mode, moves to ESP-1. It now uses I-list$_{ESP-1}$. However, it already has I-cache block addresses in I-list$_{ESP-2}$. These are copied onto I-list$_{ESP-1}$, before its head, such that the last element in I-list$_{ESP-2}$ is before the head of I-list$_{ESP-1}$. The event in normal mode of execution reads the contents of I-list$_{ESP-1}$ starting from its head, while the event in ESP-1, stores I-cache block addresses starting from an entry right after the last element copied from I-list$_{ESP-2}$.

D-list has an identical composition as I-list, except that the two circular queues are, 510 bytes for ESP-1 and 57 bytes for ESP-2.

4.4.3 Augmenting the Branch Predictor

The branch predictor modeled in our design, uses PIR to index into multiple different tables (e.g. global predictor table). We explored different design choices, as explained in Section 4.6.5, and finally settled with a separate PIR for each execution context, while sharing the rest of the branch predictor structures.

It is, however, augmented with **B-List-Direction** and **B-List-Target**. Each entry in B-List-Direction stores the instruction address of a branch as an offset from the previous list entry (4 bits), direction of the branch (1 bit) and whether it is an indirect branch (1 bit). The first two entries out of every thirty, store the retired instruction count as an offset from the last instruction count.

B-List-Target stores the branch targets of taken indirect branches. Each entry is com-

posed of the branch target stored as an offset from the instruction address in B-List-Direction (16 bits) and a bit indicating if the address could be expressed using the offset. If not, the next two entries specify the complete branch target address.

4.4.4 Miscellaneous

The cachelets do not participate in coherence, keeping any updated data values private to the speculative execution context.

Any exceptions raised during ESP modes, are silently dropped, and the execution continues ignoring them. This is fine, since the execution is speculative.

Data updates in ESP modes are only stored in the D-cachelets. If a dirty cache block gets evicted from the D-cachelets, those updated data values are lost. The speculative pre-execution, thereafter, uses the older data values present in either the L1-D cache or lower levels of the memory hierarchy. This can, potentially, lead to an incorrect path of execution, generating incorrect hints. This lowers the performance improvement possible, but does not cause correctness problems.

4.4.5 ESP for any Asynchronous Program

We have described ESP in the context of web 2.0 applications running in a web browser, which had one looper thread dequeueing and executing events from a single event queue. In the general case, there could be multiple looper threads executing events from multiple event queues. In these systems, the software runtime arbitrates between the different event queues and looper threads, and decides what event runs on which looper thread.

In such a case, each looper thread would run on a separate ESP core. When determining

the event to run on a looper thread, the software runtime, will now, additionally, predict the next two events that would run on the same looper thread. These two events, which can be from different software event queues, would execute in the ESP modes of that core. This scheme works for most events. In some infrequent cases, however, the events might not execute in the predicted order. For example, if a synchronous barrier is posted to an event queue, it would hold up processing of all subsequent synchronous tasks, while allowing later asynchronous tasks to execute ahead of them. In these cases, the information gathered for the predicted event, then must not be used. An ”incorrect prediction” bit in each entry of the hardware event queue, would regulate whether the information in I, D and B-lists should be used.

4.5 Methodology

We evaluate ESP using seven real asynchronous web applications (Table 4.1). Our benchmark has a representative application for each of the different types of web applications, ranging from e-commerce to social networking to online image editing. Each of our browsing sessions are short but complete. They capture typical user behaviors. We chose not to consider JavaScript (JS) benchmark suites such as Octane [2] and Sunspider [4], as they are shown to not resemble read world behavior of JS web applications [43].

We modified the V8 JavaScript engine in Chromium to identify JavaScript event executions within the browser's renderer process. In order to create workloads that are repeatable, we used the trace-recording component of SniperSim [11] to collect instruction traces for Chromium's renderer process running on Ubuntu 12.04. The trace captures JavaScript

Website	Actions Performed	# events executed	# inst. (millions)
amazon (amazon.com)	Search for a pair of headphones, click on one result, go to a related item	7,784	434
bing (bing.com)	Search for the term "Roger Federer", go to new results	4,858	259
cnn (cnn.com)	Click on the headline, go to world news	13,409	1,230
fb (facebook.com)	Visit own homepage, go to communities, go to pictures	9,305	2,165
gmaps (maps.google.com)	Search for two addresses, get driving, public transit and biking directions	7,298	2,722
gdocs (docs.google.com)	Open a spreadsheet, insert data, add five values	1,714	809
pixlr (pixlr.com)	Add various filters to an image uploaded from local storage	465	26

Table 4.1: Web Sites visited and actions taken, and a measure of the size of the benchmarks

Core	4-wide, 1.66 GHz OoO, 96-entry ROB, 16-entry LSQ
L1-(I,D) Cache	32 KB, 2-way, 64 B lines, 2 cycle hit latency, LRU
L2 Cache	2 MB, 16-way, 64 B lines, 21 cycle hit latency, LRU
Main Memory	4 BG DRAM, 101 cycle access latency, 12.8 GB/s bandwidth
Branch Predictor	Pentium M branch predictor, 15 cycle mispredict penalty 2k-entry Global Predictor, 256-entry iBTB, 2k-entry BTB, 256-entry Loop Branch Predictor, 4k-entry Local Predictor
Interconnect	Bus
Energy modeling	V_{dd} = 1.2 V, 32 nm

Table 4.2: Details of the architecture simulated

HW Structure	Description	ESP-1	ESP-2
L1-(I,D) Cachelet	12-way, 64 B lines, 2 cycle hit latency, LRU	5.5 KB	0.5 KB
I-list	Circular Queue	499 B	68 B
D-list	Circular Queue	510 B	57 B
B-List-Direction	Circular Queue	566 B	80 B
B-List-Target	Circular Queue	41 B	6 B
RRAT	32-entry RAT	28 B	28 B
HW Event Queue	2-entry queue	8 B	8 B
Special Registers	PC, SP, Flags, ESP-mode	12 B	12 B
All HW additions		**12.6 KB**	**1.2 KB**

Table 4.3: ESP Hardware Configuration

event execution, including the native code executed as part of library calls.

We modified SniperSim to model our ESP architecture, Runahead execution [39], and next-line (NL) prefetchers for instruction [6] and data caches. Our next-line prefetcher for the data cache is modelled similar to Intel's DCU prefetcher [21], which waits for four consecutive accesses to the same data cache line before prefetching the next. Our baseline processor (Table 4.2) is modelled similar to the one used in Samsung's Exynos 5250 ARM-based System-on-Chip (SoC). ESP architecture configuration is shown in Table 4.3. ESP adds 13.8 KB of hardware state to baseline.

We evaluated the energy expended by the different designs using McPAT 1.2 [32]. CACTI 5.3 [54] was used for the added cache-like structures.

4.6 Results

We evaluate our ESP architecture and compare it to the runahead execution [39]. We also show that it complements next-line (NL) prefetchers. Later, we analyze ESP's effectiveness in improving I-cache, D-cache and branch predictor performance separately.

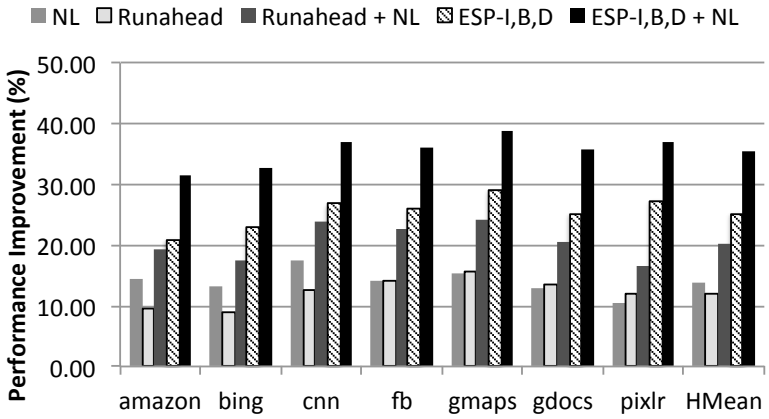


Figure 4.5: Performance of ESP, Next-Line and Runahead.

Finally, we analyze the energy overhead due to additional instructions executed in ESP.

Our ESP architecture improves I-cache (I), D-cache (D), and branch prediction (B) performance. We study the benefits due to each of these optimizations separately. The suffixes (I, D, B) that we use to ESP indicate the presence of one of the three optimizations.

4.6.1 ESP Vs Next-Line Vs Runahead

Figure 4.5 compares performance of ESP with next-line (NL) prefetching and runahead execution. Results are normalized to the baseline with no prefetching. It also shows the benefits of combining NL with ESP or Runahead execution.

We find that our ESP design with next-line prefetching can improve performance by about 35% relative to the baseline, and 19.1% compared to NL. In contrast, Runahead with next-line provides only 5.7% performance improvement over baseline.

Next-line prefetching is a simple solution, and still provides nearly 13.7% performance benefit over no prefetching. However, its effectiveness is limited as it relies on spatial locality, which is not significant in most asynchronous applications. An interesting result is that NL can complement our ESP optimization very well. For long running events, ESP

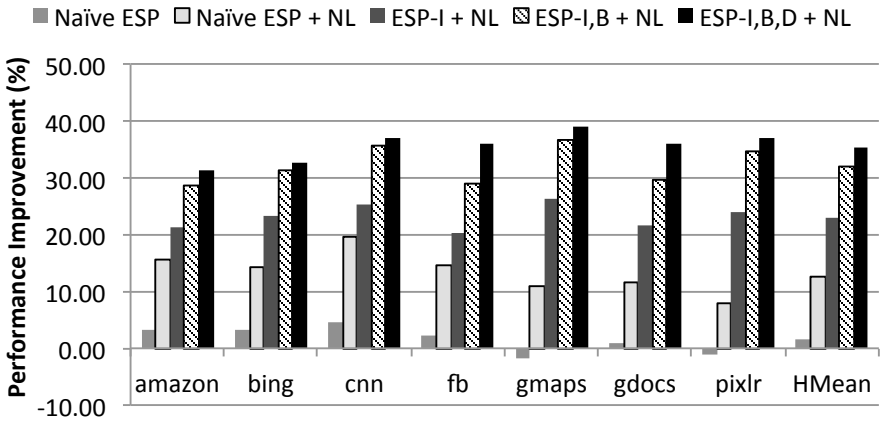


Figure 4.6: Sources of Performance in ESP.

may not have sufficient opportunity to pre-execute deep into those events. For such events, when ESP prediction is not available, simple NL performs useful prefetching. We expect that other prefetchers could similarly complement ESP.

Runahead by itself provides only 11.9% improvement over baseline. As we discussed earlier in Section 5.1, Runahead does not help improve performance in the presence of I-cache misses. As we discussed in Figure 4.2, reducing penalty due to I-cache misses is significantly more important than D-cache misses. Also, Runahead's effectiveness relies on discovering independent instructions after a load miss, which is a significant limitation. In contrast, ESP can jump ahead to the next event and easily discover independent computation to pre-execute. Nevertheless, we find that NL instruction prefetcher can complement Runahead as well as it complements ESP.

4.6.2 Sources of Performance in ESP

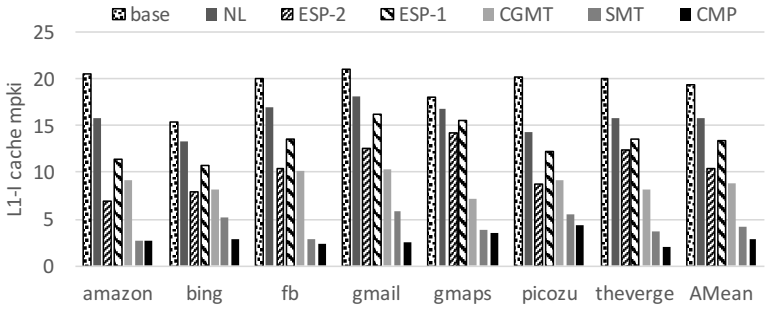
Figure 5.4 shows the sources of performance in ESP. To study the need for cachelets and predictor lists, we consider a hypothetical naive ESP design that does not use these extra structures. Instead, it fetches cache blocks into regular memory hierarchy (L1, L2)

and updates branch predictors, during event pre-executions. This design is similar in principle to Runahead designs [22, 9, 38, 37], which instantly prefetches cache blocks and updates branch predictors in the runahead mode. We find that this naive ESP design hardly improves performance. In fact, it degrades performance for some applications (`pixlr`). This is because ESP jumps much father ahead into the execution than runahead. Therefore, prefetching blocks instantly during pre-executions would be too early. Prefetched blocks may get evicted before use, and even worse, it can degrade performance by evicting useful cache blocks used by the current non-speculative event.

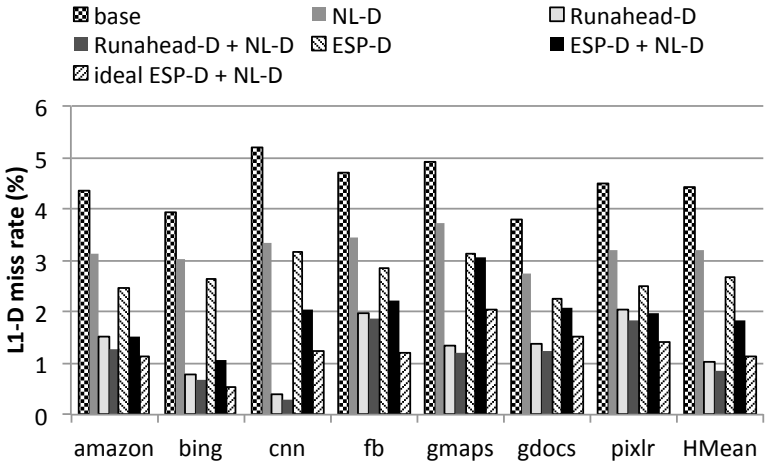
Figure 5.4 also shows that I-cache prefetches initiated through information recorded in I-lists provides the most significant benefit (13.6%). Branch prediction also provides appreciable performance gain (7.5%), but D-cache prefetching provides only marginal benefit (3.9%). This is not surprising given that the highest performance potential was by reducing I-cache misses (Figure 4.2). In the following sub-sections, we analyze each of these benefits in more depth.

4.6.3 ESP for Instruction Cache Performance

Figure 4.7a shows L1 I-cache misses per kilo-instructions (MPKI) for various configurations. NL-I represents next-line prefetching for I-cache only. ESP along with next-line reduces I-cache MPKI from about 23.5 to 11.6. To understand the potential of ideal ESP design, we evaluated an ESP configuration with infinite I-cachelet and I-list, and also assumed timely prefetches into the L1-cache. Our design comes close to ideal.



(a) I-cache performance.



(b) D-cache performance

Figure 4.7: Cache Performance

4.6.4 ESP for Data Cache Performance

Figure 4.7b shows L1 D-cache miss rate for Runahead-D, ESP-D, and their combinations with next-line data-only prefetcher (NL-D). Runahead-D is Runahead Execution but only warms up the data cache, but does not update branch predictors.

Our ESP architecture with NL-D reduces L1 D-cache miss rate from 4.4% to 1.8%, whereas Runahead with NL-D does slightly better by reducing it to 0.8%. As expected, Runahead execution performs well for D-cache performance, since it is able to warm-up the data cache in short bursts (during LLC data misses) throughout the execution of the event. ESP-D design is relatively less effective. However, an ideal ESP-D design, with infinite cachelet size and timely prefetches, performs comparably to Runahead. This demonstrates that there is room for improving ESP design further for data cache performance.

4.6.5 ESP for Branch Predictor Performance

ESP reduces branch misprediction rate from 12.9% to 7.1% (Figure 4.8). One important design question is how to update branch predictor states during pre-execution (ESP mode). Not updating the branch predictor would compromise pre-execution's ILP, whereas updating them may compromise normal event's ILP. The other problem is how to learn from the mispredictions during a pre-execution of an event so that they are prevented during its normal execution.

We carefully navigated the design space to arrive at the design discussed in Section 4.4.3. Figure 4.8 shows the branch misprediction rate for different configurations that we considered. Naively updating branch predictor state during pre-executions did not provide any

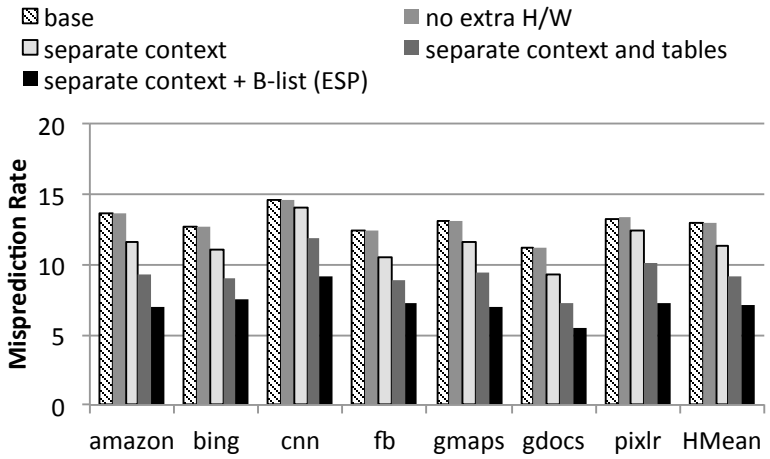


Figure 4.8: Branch misprediction rate.

gains. At another extreme, we considered replicating all the branch prediction hardware for each ESP mode ("separate context and tables"). By warming up the replicated branch predictor in the ESP mode, and using its state during the event's normal execution reduced branch misprediction rate from 12.9% to 9.2%. However, this design is clearly area inefficient.

In the ESP design, we just replicated the branch prediction "context", which is the Path Information Register (PIR) used to index into the predictor tables. Predictor tables were updated in the ESP modes. This helped us avoid significant interference between normal and the ESP modes. Coupled with the B-list that enabled us to train the branch predictor with a preset number of branches ahead of the current branch during the normal execution, our design even managed to outperform the design that replicated all branch predictor hardware (7.1% as opposed to 9.2%).

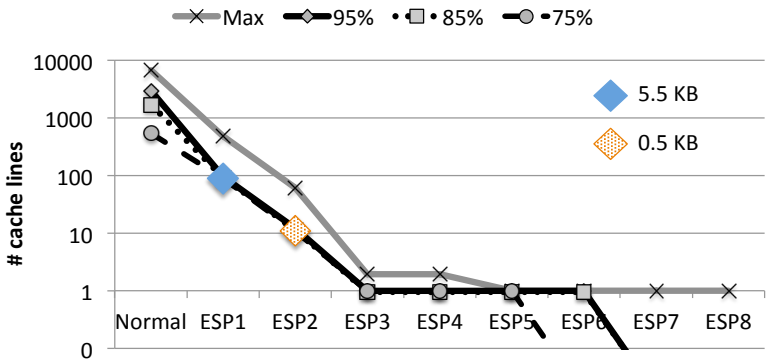


Figure 4.9: I-cachelet Size.

4.6.6 Area Overhead

We define working set of a pre-executed event to be the number of cache blocks that need to cached to exploit all reuse. Sizes of a instruction (I) or a data (D) cachelet need to be large enough to keep track of the working set of pre-executed events.

Figure 4.9 shows the max working set of all events executed in each of the different ESP modes in logarithmic scale. For comparison, it also shows the max working set of all the events executed in the normal mode ("Normal"). As pre-execution of events are relatively shorter than full normal execution, their working set sizes are order of magnitude smaller (e.g., Max of "ESP1" is 10x smaller than max of "Normal").

It is however wasteful to provision cachelets to support worst case requirements. Therefore, we estimate cachelet size required to capture 95% of reuse in 95% of all events. We find that we could further reduce cachelet size requirement by another order of magnitude to about 5.5 KB for ESP-1. Cachelet size required for higher ESP-2 mode (jump ahead two events) is even smaller (0.5 KB), as events spend much less time in this mode.

These results also show that it is wasteful to provision higher order ESP modes beyond ESP-2, because we rarely see any opportunity to touch a cache block in these modes. We

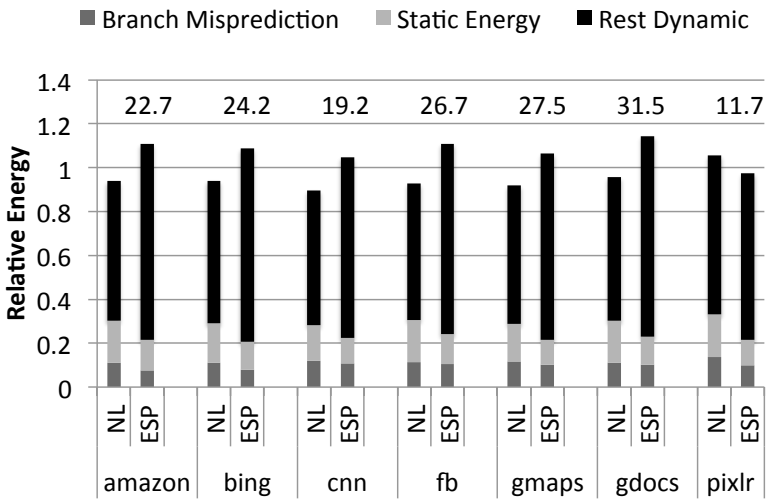


Figure 4.10: Energy Overhead.

omit results for D-cachelet size, but the conclusions were similar to I-cachelets.

Table 4.3 lists all the additional hardware structures used in ESP to support two modes. ESP-1 mode requires 12.6 KB, whereas ESP-2 mode only requires 1.2 KB.

4.6.7 Energy Overhead

ESP uses additional hardware (Table 4.3), and also executes on average 21.2% more instructions than the baseline, which can potentially increase the energy overhead. However, since it reduces latency, it can reduce static energy consumption. Branch mispredictions are also reduced, leading to less dynamic energy spent in executing wrong path instructions. Combining the above factors, ESP ends up consuming about 8% more energy than the baseline as shown in Figure 5.10. Numbers on top of the bars specify the percentage of additional instructions executed compared to the baseline.

4.7 Related Work

Unlike ESP, past work had not attempted to optimize general-purpose processor architectures for asynchronous programs. Runahead execution [22, 9, 38, 37] is perhaps the most closely related work. Sections 5.1 and 5.7 provided an in-depth comparison to this optimization.

Web Browser Optimizations: Crom [36] is a software solution that exploits event-level parallelism by speculatively executing events when the browser is idle using cloned browser context. ESP is a processor solution for hiding micro-architectural bottlenecks in asynchronous programs.

Zhu and Reddi [59] exploit slack in loading web page to save energy by executing on energy-efficient "little" cores. They also developed custom accelerators for style-resolution kernel and a specialized cache for tracking important data structures in the web browser [60]. In contrast, ESP is a more general-purpose architecture that aims to improve the performance of all asynchronous applications.

Speculative Pre-Execution: Helper threads execute just the backward slices of the program required to compute data addresses and branch outcomes to fetch cache blocks early and warm-up the branch predictor [61, 47, 20, 14, 53]. Compared to ESP, these techniques share similar limitations that we discussed for runahead, such as not being able to resolve I-cache misses.

Prefetching: In Section 5.7, we showed that next-line prefetching [6] can complement ESP. EFetch is a customized instruction prefetcher for web applications [12]. Instruction prefetchers [25, 12] in general incur significantly high hardware overhead (39 KB in

EFetch [12]), as opposed to about 14 KB in ESP. This high overhead is mandated by the relatively large instruction footprint in these applications.

ESP also has the advantage that it also helps improve branch predictor and data cache performance. While there has been significant work on data prefetching [7, 16, 17], potential for performance gains from data cache optimization is limited (Figure 4.2).

4.8 Conclusion

Conventional processor architectures have largely ignored the characteristics of asynchronous programs, which constitute a large fraction of all computing systems built today. As the processor industry embraces heterogeneity, we propose that some cores in a multi-core processor be enhanced to improve the efficiency of asynchronous program execution.

Towards this end, this chapter presented the ESP architecture. ESP peeks into the event queue to determine pending future events. It exploits this knowledge to profitably pre-execute future events in the presence of long latency cache misses. Pre-execution of an event helps ESP determine instruction and data addresses that need to be prefetched and branch mispredictions that need to be corrected. This information is later used to remove the micro-architectural roadblocks while executing an event in the normal mode.

We showed that ESP can improve the performance of seven real web applications by an average of 19%, whereas traditional runahead only achieves a 5.7% gain. We expect comparable performance improvements for other asynchronous systems such as the mobile applications with similar characteristics as the web applications.

CHAPTER V

An energy-efficient software-centric approach to accelerating asynchronous programs

Asynchronous programs exhibit unique execution characteristics due to fine-grained interleaving of short events, which destroys cache locality and branch prediction accuracy. In this work, we propose Eidetic Event Rehearsal (EER), a software-centric system that mitigates architectural bottlenecks in asynchronous programs. EER exploits the latent event-level parallelism (ELP) present in asynchronous programs. It speculatively pre-executes future events on an idle hardware thread to record useful information, that is later used to initiate accurate prefetches. Moreover, we observe that events repeat during asynchronous program execution. Exploiting this recurrence of events, EER speculatively pre-executes an event a few times, and then reuses the recorded information on subsequent dynamic instances of the same event, saving energy and system resources. We demonstrate that EER improves the performance of many popular asynchronous Web 2.0 applications including Amazon, Facebook and Google Maps by 43% on average, while saving 18% energy.

5.1 Introduction

As described earlier, despite the ubiquity of asynchronous programs, their unique execution characteristics have been largely ignored by conventional processor architectures, which have remained heavily optimized for synchronous programs. Asynchronous programs are characterized by short events executing varied tasks. This results in a large instruction footprint with little cache locality, severely degrading cache performance. Also, event execution has few repeatable patterns causing poor branch prediction.

Fortunately, asynchronous programs present unique opportunities for performance improvement. Considering Web 2.0 applications in this work, we observed that events get enqueued into an event-queue and typically reside there for thousands of micro-seconds, before being executed sequentially. The HTML 5 standard specifies sequential execution of events. However, since events perform varied tasks, they are largely independent of each other. For example, in a news web application, the event querying the device's location is independent of the one updating the latest news banner. This a priori knowledge of future events combined with their relative independence points to latent event-level parallelism (ELP) in asynchronous programs, left untapped by conventional processor architectures.

One approach to exploiting this latent ELP is to "rehearse" the execution of events residing in the event-queue before their execution. Future events can be speculatively pre-executed to record information about micro-architectural bottlenecks, like cache addresses accessed and branch outcomes. During subsequent normal execution of the event, this recorded information can be used to issue accurate prefetches and improve branch prediction accuracy, accelerating the event's execution.

The ESP architecture V used a similar approach. It employs two hardware threads in a coarse-grained multithreading (CGMT) like configuration to speculatively pre-execute the next two events in the event-queue. Hardware lists are used to store information about these micro-architectural bottlenecks. ESP uses cache-like structures (instruction and data cachelets) to isolate the working sets and data updates of the speculative pre-execution from the normal execution. Further specialized hardware extensions are used for prefetching and training the branch predictor.

While ESP provides good performance improvement of asynchronous programs, it has some noteworthy drawbacks. First, it needs extensive hardware support, makes invasive modifications to the processor pipeline and incurs hardware storage overhead (hardware lists and cachelets). Second, since extra instructions are speculatively executed for each dynamic event, it is not energy-efficient, which is of paramount importance to mobile computing. Lastly, it is not able to pre-execute deep into future events, leaving a large amount of potential performance improvement untapped.

To solve the above challenges, we propose the Eidetic Event Rehearsal (EER) system. It is composed of a subsystem (Event Rehearsal, ER) which enhances instruction cache performance, but increases energy expenditure. Eidetic Event Rehearsal encompasses ER and makes it highly energy efficient by removing redundant work.

Employing the approach of rehearsing event execution, ER speculatively pre-executes future events to improve instruction cache performance. Unlike ESP, ER is designed almost entirely in software, in an effort to not incur hardware storage overhead and reduce added hardware complexity. The little hardware support it does need is not specialized narrowly for ER, but has wide applicability. Moreover, ER uses existing multithreading mechanisms

in hardware, namely simultaneous multithreading (SMT) and chip multiprocessors (CMP).

While ER can significantly accelerate asynchronous programs, it does so by executing a large number of extra instructions during speculative pre-execution. Every dynamic event is speculatively pre-executed, making ER quite energy inefficient.

We devise a solution to this problem, exploiting an opportunity presented by asynchronous programs. We observed that events recur multiple times during asynchronous program execution. Specifically in the case of web applications, the longer the browsing session, the greater the proportion of recurring events, among all dynamic events executed. This is even more true across multiple browsing sessions. For example, composing multiple emails will lead to multiple executions of the same event.

To exploit this insight, the Eidetic Event Rehearsal (EER) system remembers its "rehearsal" of event execution. It employs ER for recording information about events through speculative pre-execution. Thereafter, it stores this information for each static event. On subsequent dynamic instances of the same events, the pre-recorded information is used like in ER, oblivious to whether this particular dynamic event was speculatively pre-executed or not. The pre-recorded information about an event can even be used across browsing sessions, further amortizing the overhead of speculative pre-execution.

Since EER obviates the need to speculatively pre-execute every future event, it can skip ahead and speculatively pre-execute events further in the future, increasing its coverage and performance gain.

This work makes important contributions in accelerating the widely used emergent class of asynchronous applications:

- Exploiting event recurrence in asynchronous Web 2.0 applications, EER accelerates them in a highly energy-efficient manner. Since this technique can be employed across multiple browsing sessions, its energy-efficiency improves with use.

- EER employs a highly software-centric design with minimal hardware support to accelerate asynchronous programs. This greatly reduces the hardware complexity and storage overhead incurred by prior work. It also makes EER more flexible and thus deployable on multiple processor configurations like SMT and CMP.

- EER is able to achieve near comprehensive coverage of future events. It can pre-execute deep into future events and thereafter, employs the normal thread to capture event information for the tail part of the event's execution. This helps eliminate most L1-I cache misses, achieving large performance gains.

- We evaluate EER on widely used real Web 2.0 applications, representing many popular uses of the web - e-commerce (`amazon`), search (`bing`), social networking (`facebook`), email (`gmail`), interactive maps (`google maps`), online image editing (`picozu`), news (`theverge`). On these applications, EER improves the performance of a baseline architecture with a next-line instruction prefetcher and Intel's data prefetchers (next-line and stride) [21] by 43%, while saving 18% energy. In contrast, ESP accelerates performance by only 15% and expends 8% more energy.

Though we demonstrate the utility of EER on Web 2.0 applications, it should be broadly applicable to other asynchronous programs.

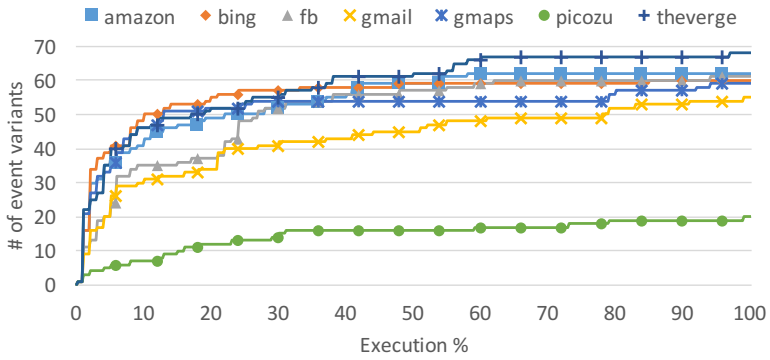


Figure 5.1: Number of unique event variants saturates with increasing session length.

5.2 Background and Motivation

In this section, we present a case for speculative pre-execution with persistent memoization, and briefly summarize relevant prior work and its limitations.

5.2.1 Event Recurrence

During asynchronous program execution, typically event handlers recur. Specifically, in the case of Web 2.0 applications, the longer the browsing session the more likely it is for event to repeat. Figure 5.1 presents evidence of this fact. It plots the cumulative number of unique event variants against execution progress for seven real web applications, where an event variant refers to a particular control flow through an event handler. Event recurrence is even more prevalent for any single user across multiple browsing sessions.

Event handlers repeat because, first, for many web applications there is a small number of unique JavaScript event handlers. Second, there are biases in a user's browsing habits, as to what parts/sections of a web application the user accesses, which limits the set of unique event handlers the user invokes. We propose to exploit this property, by speculatively pre-executing an event a few times, and using the recorded information every time it repeats during the browsing session. We go a step further, and persistently store the event

information, and use it across multiple browsing sessions. This would obviate the need to speculatively pre-execute the same event each time it repeats, saving energy.

However, the ability to speculatively pre-execute is required throughout the execution of a web application, because it is not always possible to memoize every event forever. This is because, first, these web applications change over time, introducing new events. Second, the space required to store all the recorded information may be unacceptably high, requiring the system to be selective about which events' information to preserve. The remaining events would need to be speculatively pre-executed again.

5.2.2 Prior Approaches

This section summarizes the insights of prior works and discusses their limitations that EER addresses.

ESP

To exploit ELP in Web 2.0 applications, ESP speculatively pre-executes future events. Information about micro-architectural bottlenecks thus observed is recorded and later used to accelerate normal execution of the event. ESP uses hardware structures for isolation of speculative state and memoization with subsequent utilization of event information.

Isolation of Hardware Threads: Updates from speculative pre-execution must not be visible to the normal thread to ensure correctness. ESP uses separate hardware threads in a CGMT-like configuration, which ensures isolation among the different threads in the core.

For the same reason, ESP uses an additional cache-like hardware structure (D-cachelet) to store speculative data updates from the speculative threads. The cachelet also caches

the data working set of the speculative pre-execution, keeping it from polluting the L1-D cache, helping improve performance. An I-cachelet is also similarly used by the speculative threads, for better performance through reduced interference in the L1-I cache. However, these cachelets occupy 12 KB of extra hardware space.

Event Information: The cache addresses and branch outcomes observed during speculative pre-execution are stored in separate hardware lists. The list entries also record the dynamic instruction count. Subsequently, during normal execution of the event, the current dynamic instruction count is continuously compared with the values stored in the list entries. Timely prefetches are issued to the corresponding addresses, when the difference in the two instruction counts falls below a threshold. The branch predictor is trained similarly. These lists occupy 1.8 KB of extra hardware space.

Instruction Prefetchers

Traditional instruction prefetchers are either incapable of improving I-cache performance of Web 2.0 applications or have a very high hardware overhead. This is because, these applications have very large instruction footprints with few repeatable instruction access patterns, forcing instruction prefetchers to keep track of a large amount of state or perform poorly.

5.3 Eidetic Event Rehearsal (EER): Design

In this section, we first describe our insights for exploiting event recurrence. That is followed by an overview of the main components of Eidetic Event Rehearsal (EER), and

the design choices made.

5.3.1 Exploiting Event Recurrence

As explained in section 5.2.1, in asynchronous program execution, event execution is recurrent. Specifically, in the case of web applications, the longer the browsing session, the more likely it is for events to repeat, even more so across multiple browsing sessions. In light of this, speculatively pre-executing the same event every time it is generated is inefficient. Instead, we speculatively pre-execute an event a few times, persistently store the gathered information, and later use it every time the event executes, within and across browsing sessions.

5.3.2 Design Overview

The EER system can be designed on top of multiple different underlying processor configurations, the most common among them being SMT processors and CMP. To realize the EER design, we have designed software components with minimal hardware support. These components have been designed for isolation of hardware threads, memoizing information about events, timely utilization of the information, and exploiting event recurrence. The following sections describe these in detail.

While the techniques developed by EER improve instruction and data cache performance and branch prediction, the tradeoffs between the acceleration obtained and overheads suffered for each are not the same. In our study, we observed that the overheads for improving data cache and branch predictor performance outweighed the single digit performance gains (as opposed to 62% with a perfect L1-I cache), and thus we do not seek to

improve data cache and branch predictor performance in our final design.

5.3.3 CGMT Vs SMT Vs CMP

EER, unlike ESP, uses an SMT processor or a CMP. We assume two hardware threads (or cores) in the underlying hardware, since that is the most commonly used processor configuration. The existing implementation of an SMT processor or a CMP ensures isolation of execution contexts in the core by default, which is necessary for correctness, as updates from the speculative thread must not be visible to the normal thread.

EER uses only one thread for speculative pre-execution. However, this does not result in lower performance gains (compared to two on a CGMT processor), as the speculative thread executes concurrently with the normal thread, and is active for as long as there are events to speculatively pre-execute in the event-queue. In fact, this design choice affords EER more opportunities to speculatively pre-execute. Moreover, if the speculative pre-execution of the next event finishes before the normal execution of the current event, the speculative thread moves on to events further in the event-queue.

In the SMT processor, there is obvious contention between the two threads for core resources (unlike CGMT and CMP). However, the utilization of the core by Web 2.0 applications is very low with less than a fourth of the issue width used on average. Therefore, the execution of another thread has, at best, a minimal impact on the event's normal execution.

Unlike conventional SMT designs which use throughput as the performance metric, for EER, it is the performance of the normal non-speculative thread that matters. Thus the best performing conventional policy (ICOUNT [55]) of giving fetch priority to the thread with the least number of instructions in the front-end of the core, does not result in the

best performance for EER. Since we are primarily concerned with the performance of the normal thread, we compared the above scheme with one that gives priority of fetch and issue to the normal thread. This outperformed ICOUNT, and is therefore our final design choice.

Similar contention for core resources is not a concern in the case of CMP, since the two threads run alone on separate cores. Since the speculative thread has more resources to work with on a CMP, it can make faster progress deeper into future events. This benefit comes at the cost of more hardware and increased energy expenditure.

On a CMP, the speculative core is clock-gated when not in use to save energy. Since changing power states can take tens of micro-seconds, it was not powered down. Given the short execution duration of events, the core wake-up latency could subsume the event pre-execution opportunity.

5.3.4 Data Versioning

Since data updates from the speculative pre-execution cannot be made visible to the rest of the system to ensure correctness, EER needs support for data versioning.

To solve this problem, instead of using additional hardware structures (like cachelets in ESP), EER uses space carved out from existing L1 caches in the SMT processor. This reduces available space for the normal execution and could potentially degrade performance. However, due to accurate prefetching with good coverage using information gathered during speculative pre-execution, the performance impact of reduced L1 cache space is minimal.

EER reserves a way in the L1-D cache for speculative pre-execution. Speculative stores

from the speculative thread are cached in this way. However, the speculative thread can read data from the entire L1-D cache. But, the normal thread has no access to this way. Any evictions from this way are silently dropped, including evictions of dirty blocks. This does degrade the accuracy of speculative pre-execution, but is not common, and thus, its impact on performance is minimal.

If the two threads share the L1-I cache, while there are no correctness problems with the two threads sharing it, it can lead to destructive interference, degrading the performance of the normal thread. Therefore, a way is reserved in the L1-I cache for the speculative thread for better performance through reduced interference. Both threads can read from the entire L1-I cache, but can only replace blocks from their reserved ways.

To design these speculative data requests, we introduce the concept of "coherence-less" data requests. Coherence-less data requests transfer the cache block to the reserved way of the L1 cache, but do not change the coherence state of the cache block if it is already present in the cache hierarchy. If the requested cache block needs to be brought in from the main memory, for inclusive caches a copy of it is stored in the last-level cache (L2 in our design) in the exclusive (E) state (assuming MESI protocol). All speculative data requests are designed as coherence-less data requests.

All coherence-less data read requests are allowed to be processed on a cache block in the L1-D cache. Write requests that miss in the speculative part of the L1-D cache, cause duplication of the cache block if it is already present in the non-speculative part.

On a CMP, all speculative data requests are handled similarly. However, since the speculative thread has the entire L1-D cache to itself, there is no need to reserve ways.

5.3.5 Branch Predictor

We model a Pentium M branch predictor in our baseline processor, which is shared by the two SMT threads. It tracks path history in a Path Information Register (PIR). Sharing of the branch predictor leads to pollution in branch predictor state, degrading prediction accuracy. To mitigate this, EER duplicates the small PIR for each SMT thread. We found that this design choice yields most of the benefit that can be obtained by duplicating the entire branch predictor.

On a CMP, EER uses another core for speculative pre-execution, in which case there is no contention within the core.

5.3.6 Recording Event Information

A simple way to improve L1-I cache performance through speculative pre-execution could be to warm-up the cache for future events. However, the next event might execute millions of instructions later, by which time the L1-I cache would no longer remain warmed-up.

EER instead records the I-cache miss addresses in software lists, **I-List**. To do this, we propose adding an instruction to the ISA which can query if the instruction cache block hit in the L1-I cache. The compiler inserts instructions at the start of each basic block, which store the I-cache address in the I-List if it missed in the L1-I cache.

In order to issue timely prefetches to these addresses, EER

needs to be able to associate miss addresses with execution progress. Unlike a hardware mechanism, which can keep track of an execution progress metric (e.g. dynamic instruc-

tion count) and compare with recorded list entries frequently, with almost no performance overhead, a software scheme can check for execution progress far less frequently to keep performance overheads low. To aid in this, the I-List also stores markers for identifying function calls and returns, inline with the miss addresses. Timely prefetches to these addresses are then made using these markers when this event executes in the normal mode.

Even though the speculative thread populates the I-List, stores to the I-List need to be non-speculative, distinct from all other speculative stores issued by the speculative thread. Moreover, since entries in the I-List will not be used by the speculative thread or by the normal thread until the corresponding event is about to begin normal execution, the cache blocks which store the I-List do not need to be in the L1-D cache, where they would displace other useful cache blocks. Thus, stores to the I-List are also marked as non-temporal with respect to the L1 (not cached in the L1), but the corresponding cache blocks are allowed to reside in the last-level cache.

To achieve this, we propose adding instructions to the ISA that allow the software to specify stores as non-temporal with respect to a specific level of the memory hierarchy. Prior work [40] has shown that such instructions have much wider applicability than EER, even though they are not supported by current commodity processors.

For EER on CMP, while such stores still need to be distinguished as non-speculative, they are not marked as non-temporal. This is because, preserving L1-D cache capacity is less of a concern as the speculative thread has the entire L1-D to itself. Also, being able to cache the data in the L1-D, even though it is never read by that core, is more energy efficient.

Executing extra instructions to record L1-I cache miss addresses incurs some perfor-

mance overhead. Thus, as far as possible, it is not desirable for the normal thread to execute these extra instructions. Thus, the compiler also inserts a branch condition checking a boolean flag (`if_record`) guarding the execution of these instructions. This flag is set to true for speculative pre-execution, but false for the large majority of normal event execution.

Speculative pre-execution of an event might not complete before the normal thread starts execution of this event. In such a case, since speculative pre-execution must remain ahead of normal execution, it abandons pre-execution of this event, and starts pre-executing the next event. Thus, the I-List is left incomplete, which is clearly undesirable, as future instances of this event will not be able to benefit from prefetching for the entirety of their execution lengths. Thus, in such a scenario during the normal execution of an event, when the I-List runs out, compiler inserted code sets the `if_record` flag to true, enabling the normal execution to complete the I-List.

5.3.7 Mitigating Micro-architectural Bottlenecks

Avoiding a complex custom hardware prefetching mechanism, EER uses software instructions to prefetch cache blocks into the L1-I cache, during normal execution. Similar to the already supported data prefetching instructions, we propose adding an I-cache block prefetching instruction to the ISA.

Using the recorded L1-I cache miss addresses (I-List entries), EER launches timely prefetches. The I-cache blocks must not be prefetched too early, lest they be evicted before being used; too late, and at least part of the memory latency gets exposed. Also, since EER uses software instructions, mitigating the overhead of these instructions is an

important concern.

To prefetch, EER first needs to read the I-List entry (miss address) and then launch a prefetch to the address just read. Since both of these memory requests can miss in the last-level cache, these instructions need to be executed enough in advance of the access to the I-cache block, so as to be able to hide the latency of two back-to-back main memory reads. To do so, it is sufficient to insert these instructions at program points that regularly occur during program execution (e.g. branches), and prefetch for L1-I cache misses that are likely to occur after many such points in the future.

However, executing these extra instructions at every branch would be prohibitively expensive, and unnecessary since

branches occur far more frequently than L1-I cache misses. Function calls and returns, on the other hand, occur much less frequently. Also, since these involve program control to typically jump to an address beyond the current I-cache block, L1-I cache misses are more likely to happen right after function calls and returns, and by extension, between any two function calls and returns. In light of this, in our design the compiler inserts the instructions to read the I-cache miss address from the I-List and prefetch it, immediately after function call and return sites (prefetch sites). Using the markers identifying these prefetch sites in the I-List, at every prefetch site, EER launches prefetches for I-cache blocks that had missed a preset number (six) of prefetch sites ahead of the current point in program execution, during speculative pre-execution.

We also insert instructions to start prefetching from the I-List, in a function inside Chromium [1] (an open-source browser) such that these instructions execute approximately a thousand instructions before the start of an event. This early start of prefetching helps it

to stay ahead of instruction cache block accesses.

As long as there are addresses in the I-List for the currently executing event, prefetches are issued to those addresses. However, speculative pre-execution is not able to pre-execute every event in its entirety. Thus, when the addresses in the I-List run out, no more software prefetches are issued and the baseline prefetcher is the only prefetcher left active. This way, EER is complemented by the baseline prefetcher.

5.4 Event Variant Prediction

In web applications, the same events repeat execution multiple times because the user performs the same actions again or revisits the same web page or different web pages share JavaScript programs. The recurrence of events becomes more likely with longer browsing sessions, and with increasing number of sessions. This property affords an opportunity to make EER significantly more energy efficient.

Instead of speculatively pre-executing each upcoming event to gather information about it, EER speculatively pre-executes an event, records the L1-I cache miss addresses seen in the I-List, and persistently stores this event's I-List associating it with a unique event-ID using a hash map. When the same event is enqueued again, there would be no need to speculatively pre-execute it, since EER can simply use the pre-recorded information. This not only saves a significant amount of energy, but also affords EER more opportunity to speculatively pre-execute other events further in the future whose I-List may not exist yet. This also enables EER to pre-execute more events to a greater extent.

However, there can be different control flow paths through an event, leading to different

L1-I cache misses across different dynamic instances of an event. Thus, using one I-List per event is not sufficient. If the sets of L1-I cache miss addresses of two dynamic instances of an event (identified by its event-ID) differ by more than a threshold amount, we designate them as two event variants of the same event. To capture this event variance, instead of using one I-List per event, EER records I-Lists of all event variants of each event.

On coming across an event with an existing I-List record, EER needs to select an event variant whose I-List would be used for prefetching. It uses a predictor to predict the right event variant that the event would follow in its upcoming execution. We observed that the event variants of an event occurred in repeating patterns. The intuition behind this is that the history of control flow paths taken through an event modify the state of the event. This can have a significant impact on the future control flow, as there are many data dependent branches present inside an event body. This is similar to a branch whose outcome depends on its local history.

Inspired by this, we used a design similar to a simplified local branch predictor for our *event variant predictor*. The local history here consists of the event variants that the event executed the last n times. For each event, EER maintains a hash map indexed by the local history pattern of that event, with each entry containing the predicted event variant.

Before using the event variant predictor, it is trained for a sufficient number of events. During this training period, the recorded I-List after the end of the normal execution of an event is compared with I-Lists of all existing event variants of that event to determine which event variant was just executed. If it is a new event variant, the I-List is stored. The local history and the event variant predictor is updated with the event variant number of the past event execution. The training period ends when the accuracy and coverage of the

predictor rise above a threshold value, at which point EER has sufficient confidence in the predictions.

Thereafter, for every imminent event, the event variant predictor is used to make a prediction. It can do so, if the event has executed before, the event variant history has been seen before and the I-List for that event variant exists. If the above conditions are true, the pre-recorded I-List for that specific event variant is used for prefetching. If no prediction can be made, the same process as in training is followed.

While the event variant prediction is highly accurate, sometimes it does predict incorrectly, or the event exhibits an, as yet, unseen event variant. This leads EER to prefetch I-cache blocks from the I-List of a different event variant. While this is undesirable, it still helps improve performance, since event variants are not highly dissimilar. Even so, it is important to recognize the misprediction and train the predictor accordingly, to improve prediction accuracy in the future.

This is non-trivial, as we do not know the right event variant, because this dynamic instance of the event was not speculatively pre-executed. To solve this, EER compares the number of L1-I cache MPKI (misses per kilo-instructions) incurred during the past event execution (obtained through hardware counters) with a preset threshold. If this number is higher than the threshold value, it is considered a misprediction. The corresponding entry in the event variant predictor for this local history is removed. This gives EER the opportunity to speculatively pre-execute the event the next time this event variant history is observed, possibly persistently store the I-List if it proves to be a new event variant and update the event variant predictor.

We also consider the case where the space needed to store the I-Lists is limited. In

which case EER needs to selectively discard I-Lists of certain event variants. Ideally, we would want to discard those event variants, which are the least used in the future. Assuming that history is reflective of the future, EER employs the following policy - when the size of the stored state crosses the limit, the least frequently used event variant is discarded, favouring the most recently seen event variant to break ties.

5.5 Implementation Details

In this section, we describe the details of the different software and hardware changes made to realize EER.

5.5.1 Isolating Speculative Cache State

To isolate speculative data updates on an SMT processor, EER reserves a way in the L1-D cache. Also, all data requests from the speculative thread, except stores to the I-List, are marked speculative which can be implemented using coherence-less data requests (as described in Section 5.3.4). A way reserved in the L1-I cache for speculative pre-execution helps in mitigating the performance degrading effects of destructive interference.

While the performance loss of the normal event execution due to reduced cache capacity is minimal because of accurate and comprehensive prefetching, it can be further mitigated. When speculative pre-execution is inactive, if pre-recorded information is due to be used for all current events in the event-queue, we can take advantage of the entire L1 cache. To this end, we propose adding an instruction that can reserve/release ways in the cache. When releasing ways in the L1-D cache used by the speculative thread, all blocks in these

ways are invalidated first.

5.5.2 Memoizing I-Cache Addresses

Since we cannot simply warm-up the caches during speculative pre-execution, EER memoizes the L1-I cache miss addresses in a software data structure per event variant called the I-List. These addresses are later prefetched during normal execution.

During speculative pre-execution, at the start of every basic block, the compiler inserts instructions to check if the I-cache block missed in the L1-I cache. If so, the I-cache block address is stored in the I-List. Markers indicating a function call and return are also stored along with the addresses.

To query for L1-I cache misses, we propose adding a condition flag in the processor which is set on an L1-I cache miss and reset on a hit, called the *instruction cache miss* (`im`) flag, like the zero (`z`) flag in x86. Also, we propose extending the family of conditional branch instructions to include a check to this condition flag called "jump on instruction cache miss", `jim` (similar to "jump on zero", `jz`, in x86 ISA).

Each entry in the I-List is a record with up to three fields. The first field is 2 bits long and specifies the type of the record, which is one of four different types - compacted address, full address, function call, function return.

If it is a compacted address, the next 16 bits represent the cache block address as an offset from the previous cache block address. The 6 bits after that represent a bit mask. These bits correspond to the next 6 contiguous cache block addresses. A 1 implies the corresponding cache block missed in the L1-I cache, a 0 implies a hit. This helps represent the next 6 cache block addresses compactly.

If it is a full address, the next 26 bits represent the complete cache block address (for 32 bit address with 64 byte cache blocks). The 6 bits after that represent a bit mask used for the same purpose as above. This is used if the cache block address cannot be represented as a compacted address.

The remaining two types act as markers representing a function call and return. These record types only have this field.

To keep the speculative pre-execution ahead of normal execution, an interrupt is generated at the end of an event which compares the event number of the next event with the one currently being speculatively pre-executed. If they are equal, it means the normal thread has caught up, and the speculative thread terminates its current execution moving onto pre-executing the next event.

5.5.3 Prefetching I-Cache Blocks

The entries stored in the I-List are used to launch timely prefetches. During normal execution instructions are inserted at the start of basic blocks following function calls and returns, that read the records in I-List and launch prefetches, using an instruction prefetching instruction, `prefetchiht0`, that we propose to add to the ISA.

Records up to the next function call or return are read, and any addresses read are prefetched. It is possible that no address record exists between two function calls or returns, in which case no prefetches are made.

Web Site	# of ins. (billions)	# of events executed	# of ins. after training (billions)	I-Queue size, SMT (KB)
amazon (amazon.com)	3.4	1,760	0.86	683
bing (bing.com)	3.6	2,240	0.98	723
fb (facebook.com)	2.4	680	0.34	756
gmail (gmail.com)	3.0	1,040	0.20	672
gmaps (maps.google.com)	3.5	1,200	0.69	918
picozu (picozu.com)	0.2	760	0.06	86
theverge (theverge.com)	3.0	800	0.94	998

Figure 5.2: Benchmarked Web Applications.

5.5.4 Event Variant Predictor

To exploit event recurrence, EER stores the I-List for each event, and does not discard it after using it. Each event is uniquely identified by an event-ID, which is used to index into a hash map which points to I-Lists of events.

EER uses a local predictor to predict the next event variant of an event, with a local history of 10 past event variants. This value was chosen through empirical analysis, described in Section 5.7.4. Using the event-ID and the predicted event variant, EER is able to use the right I-List for prefetching.

5.6 Methodology

We have evaluated EER on seven real world asynchronous Web 2.0 applications (Figure 5.2). These range from e-commerce and email to online image editing and online interac-

tive maps. These popular web applications represent some of the most common uses of the web today. We chose these web applications instead of JavaScript benchmark suites like Octane [2] and Sunspider [4], since the latter have been shown to not represent real world usage of JavaScript [43].

We instrumented Chromium v44 [1] running on Ubuntu 14.04, to demarcate JavaScript event execution boundaries. Since an interplay between multiple asynchronous input sources (user, network) is involved, web application execution is non-deterministic. To get repeatability across multiple evaluations, we used instruction traces of the renderer process of Chromium, recorded while browsing the chosen web applications. The trace recording component of Sniper [11] was used to record these instruction traces. While recording traces, the number of events present in the event-queue are measured at the start of every event. Speculative pre-execution is restricted from progressing any further than this recorded number of events.

To validate our claim that events are largely independent of each other, we modified Chromium such that before the execution of an event, we would start executing the next two events in the event-queue in separate forked-off renderer processes. The current event would then execute concurrently with the next two events. The instruction traces of these future events thus recorded, were used to simulate speculative pre-execution of events.

More than 98% of such speculative events, executed to completion. Comparing the speculative pre-execution of events with their subsequent normal executions, we observed that they were more than 99% similar, showing that most events are largely independent of each other. The remaining events failed when they branched differently than the non-speculative execution.

Core	4-wide, 1.66 GHz OoO, 96-entry ROB, 16-entry LSQ 32-entry issue queue
Branch Predictor	Pentium M branch predictor 15 cycle mispredict penalty 2k-entry Global Predictor, 256-entry iBTB, 2k-entry BTB, 256-entry Loop Branch Predictor, 4k-entry Local Predictor
Prefetchers	Instruction: Next-line (NL); Data: NL, Stride (256 entries)
Multi-threaded	SMT: 2-way, 1 active thread
Memory Subsystem	2-level inclusive caches, MESI coherence protocol
L1-(I,D)-Cache	Private 32 KB, 4-way, 64 B lines, 2 cycle hit latency, LRU
L2 Cache	Shared 2 MB/core, 16-way, 64 B lines, 21 cycle hit latency, LRU
Interconnect	Bus
Main Memory	4 GB DRAM, 101 cycle access latency, 12.8 GB/s bandwidth
Energy	V_{dd} = 1.2 V, 32 nm technology node

Figure 5.3: Baseline Architecture Modeled.

The different architectures and designs were modeled and evaluated using Sniper. This includes the baseline processor (Figure 5.3), modeled after the processor in Samsung mobile Exynos 5250 ARM-based SoC, the ESP architecture [13], next-line (NL) prefetchers and our EER design. The NL prefetcher for data is based on Intel's DCU prefetcher [21], which waits for four consecutive accesses to the same data cache line before prefetching the next.

The energy expended was modeled using McPat 1.0 [**?**].

5.7 Results

In this section, we evaluate our EER system, and compare it against ESP [13] and next-line (NL) prefetching. EER can be used over multiple underlying hardware substrates, and we compare and analyze the performance of EER on an SMT processor and a CMP.

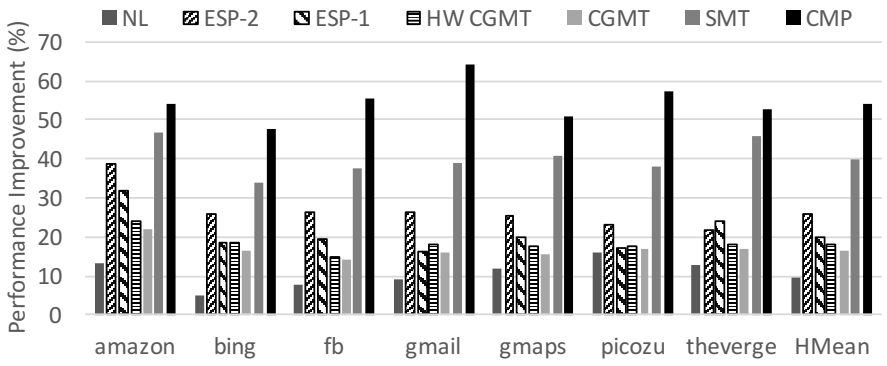


Figure 5.4: Performance of Next-Line, ESP and ER.

Thereafter, we analyze the mechanism and benefits of exploiting event recurrence, followed by a study of the overheads incurred and performance of multiple hardware instruction prefetchers.

We refer to the non-eidetic component of EER (ER) deployed on CGMT and SMT processors and CMP solely by the mnemonics of these processor configurations. Our final design (EER) is indicated similarly, by using the prefix E.

5.7.1 ER Vs ESP Vs Next-Line

Figure 5.4 compares the performance of ER with ESP and next-line prefetching. Performance improvement is calculated with respect to the baseline of no prefetching. Data stride (S) prefetching provides an almost insignificant performance benefit of 0.1% over next-line for these workloads, and has not been shown separately. All designs include the baseline prefetcher (next-line).

ER on SMT improves performance by about 40% over the baseline, and 28% over NL, whereas ESP gets only 26% over the baseline. ER on CMP further increases the performance improvement to 54% over the baseline.

Next-line prefetching is a simple design that seeks to exploit spatial locality in the ad-

dress stream. It gets almost 10% performance improvement over the baseline. It does not do much better since there is not a lot of spatial locality in the instruction address stream, as JavaScript events in web applications exhibit complex control flow with many branches. Also, any improvement in data cache performance translates to a small performance improvement.

ESP uses three hardware threads, one for normal execution and two for speculative pre-execution of the next two events, in a CGMT-like configuration. We designate this design as ESP-2 (two speculative threads). Since ER uses one speculative thread, we also compare it against an ESP design with one speculative thread (ESP-1), which suffers from reduced opportunity for speculative pre-execution.

Unlike ER, ESP memoizes all event information in hardware lists, and uses hardware mechanisms to prefetch cache blocks and to train the branch predictor. Thus, at the cost of increased hardware complexity and storage, it does not incur any performance overhead for memoization, prefetching and improving branch prediction. More speculative pre-execution increases performance improvement without incurring any performance overheads, which is why ESP-1 performs worse than ESP-2.

Despite incurring almost no performance overhead, ESP performs worse than ER on SMT and CMP, since it gets less opportunity to pre-execute future events. This is due to, firstly, its CGMT-like configuration. Unlike threads in SMT processors and CMP, speculative threads on ESP only run when the normal thread is stalled.

Secondly, in ESP, once the speculative pre-execution of the next event, or the one after that, completes in its respective thread, the thread remains idle for as long as the normal thread takes to complete execution of the current event. This is a missed opportunity, espe-

cially in the case of many short events followed by a long event, where multiple imminent events could have been pre-executed, but are not. On the other hand, the speculative thread in ER keeps moving on to the next event in the event-queue, on completion of an imminent event.

To analyze the performance impact of the above differences, we evaluated a hypothetical design, HW CGMT. This uses ER on a 2-way CGMT processor, but it does not incur the performance overhead of software memoization and prefetching instructions. Due to the second reason above, HW CGMT has a lower L1-I cache miss rate than ESP-1 (Figure 5.5). Despite this, HW CGMT performs worse than ESP-1 because, unlike HW CGMT, ESP-1 also benefits from improved L1-D cache performance and branch prediction.

ER on CGMT performs slightly worse than HW CGMT, reflecting the performance overhead of memoization and

prefetching using software instructions. ER on SMT and CMP provide progressively higher performance due to the increased proportion of future events they are able to speculatively pre-execute. This fact is borne out in the number of extra instructions executed during speculative pre-execution by the different designs (Section 5.7.6).

5.7.2 Instruction Cache Performance

Figure 5.5 shows the L1-I cache misses per kilo instructions (MPKI) for different configurations. ER reduces the baseline L1-I cache MPKI from 19.3 to 4.2 on an SMT processor and 2.9 on a CMP. The lower L1-I cache MPKI of CGMT (8.9) compared to ESP-1 (13.3), seems contradictory to its lower performance. As explained earlier, this is because ESP-1 also benefits from improved branch prediction and L1-D cache performance.

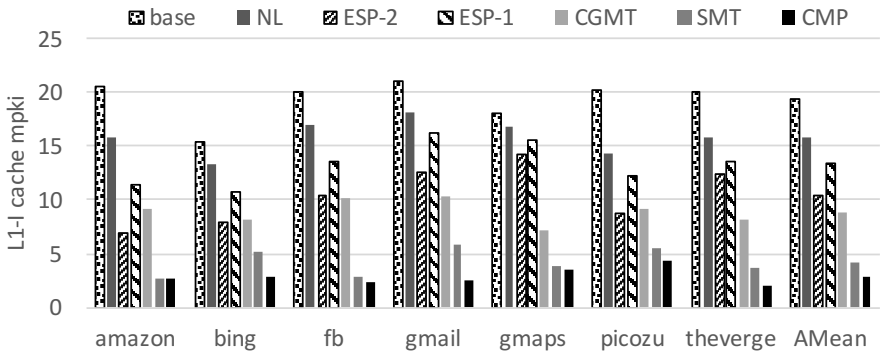


Figure 5.5: L1-I cache performance.

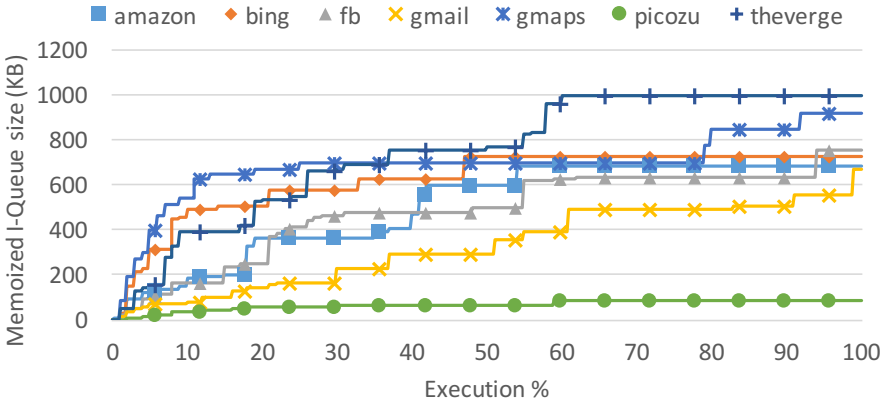


Figure 5.6: Cumulative size of memoized I-List gradually flattens out.

5.7.3 Event Variance

To record the misses incurred by different control flow paths through an event, EER records an I-List for each event variant. This is important, since event variants of an event can differ significantly. Using the proportion of L1-I cache misses that are common across two event variants, Figure 5.7 shows that about 38% of event variants match each other by less than

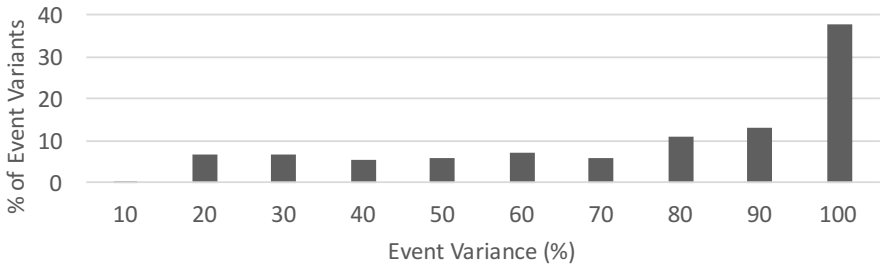


Figure 5.7: Histogram of the percentage of event variants, binned by how similar they are to each other.

10%.

Recording an I-List per event variant does increase the size of information that is memoized, but still does not become intractable, since the number of event variants per event is very limited. We observed that more than 95% of the execution is covered by a maximum of 4 event variants per event, and no event had more than 5 event variants.

Figure 5.1 shows the cumulative number of event variants observed as the execution progresses. Web applications like e-commerce (`amazon`) and the news (`theverge`), show a higher number of event variants than the rest, because of the large number of scripts executed by them. The cumulative size of all I-Lists that are memoized for a web application shows a similar trend (Figure 5.6).

5.7.4 Event Variant Prediction

To be able to reuse the pre-recorded I-List without speculative pre-execution, EER needs to predict the event variant that the next dynamic instance of the event will exhibit. EER uses a predictor similar to a local branch predictor. We studied the performance of the event variant predictor using different history lengths, and evaluated it using two metrics. The predictor makes a valid prediction only if it has seen the current local history pattern before. *Coverage* of the predictor is the percentage of times it can make a valid prediction. *Accuracy* of the predictor is the percentage of correct predictions when it can make a valid prediction.

Achieving high values for both metrics is desirable. Through empirical analysis, we observed that increasing the history length reduces coverage (as more patterns need to be learnt) and increases accuracy (there is less aliasing between patterns). Using a history

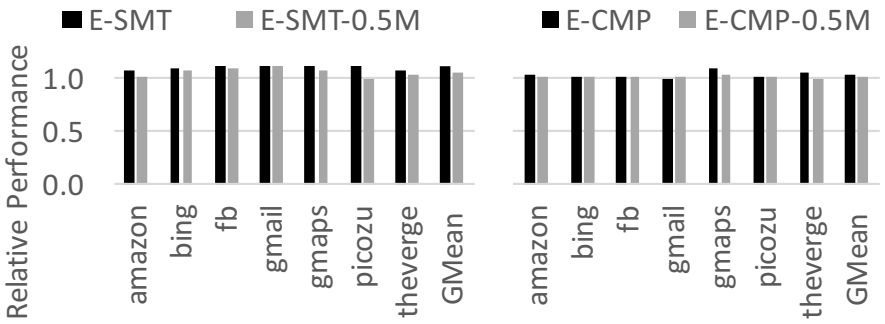


Figure 5.8: Performance relative to ER on SMT (left) and ER on CMP (right). EER and its space constrained version show increased performance relative to ER.

length of 10 leads to a good tradeoff between accuracy and coverage.

5.7.5 Employing Eidetic Memory of EER

Using the event variant predictor, EER tries to predict the event variant for the next event. If it can make a valid prediction, the corresponding pre-recorded I-List is used for prefetching. This saves EER from having to speculatively pre-execute the event.

As a result, the speculative thread can speculatively pre-execute more events further in the future, if any are available. If not, the speculative thread remains stalled leaving more resources for the execution of the normal thread, potentially improving performance.

Also, pre-recorded I-Lists are complete (with help from the normal thread), which is not always true for an I-List just recorded through speculative pre-execution. Thus, using a pre-recorded I-List can lead to better performance.

On the other hand, the prediction could be incorrect, which would lead to subpar prefetching accuracy and coverage, degrading performance.

The cumulative effect of the above factors is reflected in the performance shown in Figure 5.8. E-SMT and E-CMP refer to EER on SMT and CMP, respectively. The performance of the left sub-graph is relative to ER on SMT, and for the right sub-graph to ER on

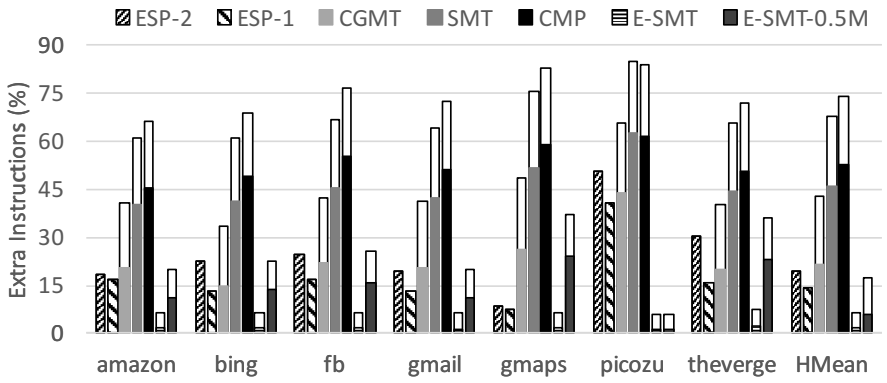


Figure 5.9: The shaded bars display the percentage of extra instructions executed due to speculative pre-execution with memoization in hardware. The white bars stacked on top of them show the percentage of software instructions executed for memoization and prefetching. The baseline is the number of instructions executed by the normal thread, with no software prefetching.

CMP. In most cases, EER outperforms ER. The performance difference is higher in the case of SMT, since the I-Lists recorded solely through speculative pre-execution on SMT are less complete compared to those on CMP.

We also consider the case where there is a limit imposed on the total size of all the I-Lists. When the size of the I-Lists exceeds this limit, the I-Lists of some event variants are selectively dropped. These will need to be speculatively pre-executed on their subsequent executions. Using 0.5 MB as the limit, we see that this configuration still outperforms ER on average, but slightly losing out to EER with no space limitations.

5.7.6 Extra Instructions and Energy Expended

The extra instructions executed can be divided into two categories, as shown in Figure 5.9. It is evident that ER on CMP is able to pre-execute the furthest into future events. This is because the speculative thread is on a separate core, and thus has more core and cache resources helping it to execute faster than other designs.

It is interesting to look at E-SMT and E-SMT-0.5M. Even though they execute only

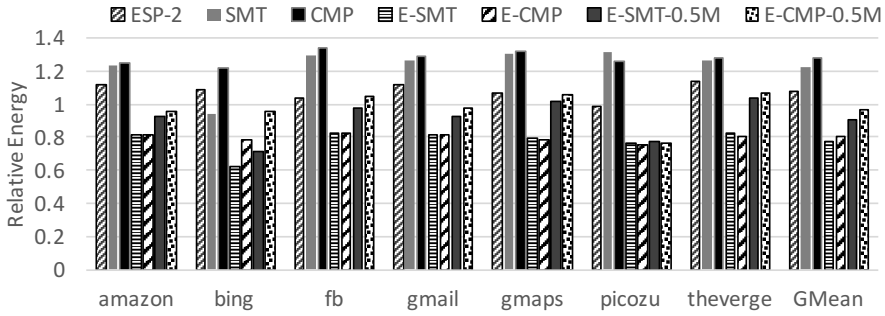


Figure 5.10: Energy expended relative to the baseline architecture.

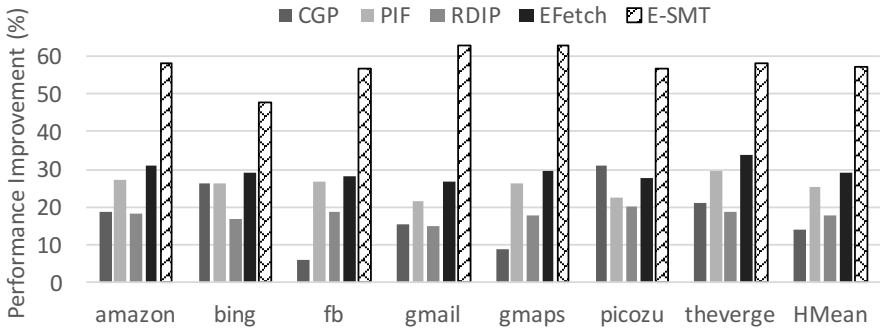


Figure 5.11: Performance of prior instruction prefetcher designs.

6% and 17% extra instructions, respectively, compared to 68% extra instructions executed by ER on SMT, they outperform it. Avoiding redundant pre-execution of events makes EER an extremely energy efficient design (Figure 5.10), saving 22% energy over the baseline. EER on CMP loses some of the energy benefit due to more static energy expended on the extra core and cache resources. E-SMT-0.5M and E-CMP-0.5M lose some of their energy benefits due to repeated speculative pre-execution of discarded event variants.

5.7.7 EER vs Alternative Approaches

Performance: Asynchronous Web 2.0 workloads have large instruction footprints with few repeatable instruction access patterns. This leads to poor performance of traditional instruction prefetchers or forces them to keep track of a large amount of state. Figure 5.11 shows the performance of many high performing instruction prefetchers - EFetch [12], PIF [25], RDIP [31], CGP [8] and E-SMT. EER clearly outperforms these designs.

Cost: The hardware storage overhead of the above instruction prefetchers ranges from 32 KB - 204 KB [12], while ESP incurs an overhead of 14 KB. EER outperforms all of these designs with almost no hardware storage overhead. However, by using an extra hardware thread it trades off computation time for storage overhead. This cost is minimal for two reasons. First, the extra hardware thread is used only during the short initial learning phase of EER. Second, there is no dearth of idle hardware threads in current processors.

5.8 Related Work

Unlike EER, past work has rarely looked to improve the performance of asynchronous programs on a general-purpose processor. ESP [13] is probably the most closely related work. Similar to EER, ESP could also be employed on an SMT processor, which would give it a performance boost from increased coverage of future events, but it would also increase its hardware storage overhead in recording more event information. However, ESP still does not address the primary contributions of EER. Further in depth comparisons to ESP have been discussed in Sections 5.1, 5.3 and 5.7.

Speculative Pre-Execution: Backward slices of branches that are difficult to predict and loads that miss in the cache often are executed speculatively on different threads. This thread executes ahead of the normal execution, and warms-up the cache and branch predictor for it [14, 53, 61, 47, 20, 19]. Runahead execution [22, 9, 38, 37] uses the same insights, but speculatively executes future independent instructions during long pipeline stalls of the normal thread. Unlike EER, the above techniques cannot mitigate instruction cache misses. Also, they require extensive hardware support, while EER is designed pri-

marily in software. However, since the future events that EER pre-executes might not start millions of instructions after the current event, EER cannot simply warm-up the caches and train the branch predictor. Thus novel solutions needed to be designed.

Optimistic Concurrency: Events in asynchronous programs could be executed concurrently, optimistically assuming they are independent, which is verified when an event finishes execution. If the events prove to be independent, the work done can be committed. However, this would require extensive hardware support - extra hardware threads, versioning, a mechanism to check for dependencies between concurrent events and to recover from mis-speculations (e.g. TLS [49, 27, 52, 35], TM [29]). Furthermore, it has been observed that even though events are largely independent of each other, there are low level memory dependencies between most event pairs. Thus, naïvely checking for memory dependency conflicts would be too conservative and might lead to serial execution of most events.

Prefetching: EER can be complemented by hardware prefetchers, as explained in Section 5.3.7. However, by themselves, conventional instruction prefetchers perform poorly, as explained in Sections **??**, 5.2.2 and 5.7.7.

Web Browser Optimizations: Crom [36] exploits event-level parallelism by speculatively executing events when the browser is idle using cloned browser contexts. EER is a software solution for mitigating microarchitectural bottlenecks in asynchronous programs, with minimal hardware support.

Zhu and Reddi [59] save energy by exploiting slack in web page load time by executing on energy-efficient cores. They similarly developed an event scheduler [58] which saves energy by exploiting slack in event execution latency. They also developed custom accelerators for style-resolution kernel and a specialized cache for tracking important data

structures in the web browser [60]. In contrast, EER is a more general-purpose system designed to improve the performance and energy-efficiency of asynchronous applications.

Domain-Specific Accelerators: Like EER, some previous works have identified the opportunities provided by event-driven sensor network applications and have designed energy-efficient systems [28, 23] for these applications. However, unlike EER, these techniques have designed hardware accelerators, specialized for one domain of asynchronous programs (sensor networks) and only improve energy-efficiency. EER can be used to improve performance and energy of a wide range of asynchronous programs including sensor networks.

Trace Cache: There is some similarity between different event variants and traces in a trace cache[46]. However, traces in a trace cache (tens of instructions) are orders of magnitude shorter than event variants (millions of instructions). Thus, predicting event variants requires using more high level information than the address of the first instruction and branch predictions.

5.9 Conclusion

Asynchronous programs have unique execution characteristics, and thus present new performance challenges. However, despite the ubiquity of asynchronous programs, conventional processor architectures have largely ignored them. With the widespread use of multi-threaded and multi-core processors, and not enough parallelism in the present applications to exploit them, we propose that some of these idle hardware threads be used to improve the performance and energy-efficiency of asynchronous program execution.

To this end, this paper presents EER. It exploits the latent event-level parallelism of asynchronous programs to accelerate them. Using primarily software systems, with minimal hardware support, it speculatively pre-executes imminent events, records the L1-I cache miss addresses, which are later prefetched, accelerating the normal execution of the event. Moreover, exploiting the recurrence of events in asynchronous program execution, EER pre-executes an event a few times. Subsequent dynamic instances of the same event are not pre-executed, and use the pre-recorded information instead.

This way, on seven real Web 2.0 applications, EER achieves a performance improvement of 43% while executing only 6% more instructions, and saving 18% energy.

CHAPTER VI

Conclusion

Asynchronous programming has become the dominant programming model. All the Web 2.0 JavaScript applications (e.g., Gmail, Facebook) use asynchronous programming. There are now more than two million mobile applications available between the Apple App Store and Google Play, which are all written using asynchronous programming. Distributed servers (e.g., Twitter, LinkedIn, PayPal) built using actor-based languages (e.g., Scala) and platforms such as `node.js` rely on asynchronous events for scalable communication. Internet-of-Things (IoT), embedded systems, sensor networks, desktop GUI applications, etc., all rely on the asynchronous programming model.

In spite of the ubiquitous use of asynchronous programming, today's general-purpose processor architectures are heavily optimized for the synchronous programming model, and ignores the unique execution characteristics of asynchronous programs.

Contributions: This dissertation looks at novel opportunities presented by the asynchronous programming model and designs processor optimizations to increase the performance of events in Web 2.0 applications (a case study of asynchronous programs). Asynchronous programs, especially web applications, tend to exercise large instruction

footprints to support a rich set of features, resulting in significantly higher instruction cache miss rates than synchronous programs. L1 instruction cache misses significantly degrade performance. On average, across the web applications studied, performance can be increased by 53% if all L1-I cache misses are eliminated. We propose an instruction prefetcher, EFetch, customized for asynchronous programs to help alleviate the performance penalty due to L1-I cache misses. For a set of real-world popular asynchronous Web 2.0 applications including Amazon, Google maps, and Facebook, we show that EFetch outperforms the commonly implemented next-2-line prefetcher by 17%.

Despite the performance challenges of asynchronous programs, they also present unique opportunities. In an asynchronous program, events are enqueued in an "event queue" before they are processed sequentially by a thread. By exposing the event queue to the processor, the processor can gather information about them ahead of time. This information can then be used to significantly improve the accuracy of various hardware predictors when an event is executed. We have designed the ESP architecture that takes advantage of the above insight to accelerate events' execution. For the same set of Web 2.0 applications as above, ESP improves performance by an average of 19%.

While ESP provides substantial performance improvement, it incurs an energy overhead and requires complex and expensive hardware support. EER exploits the latent event-level parallelism (ELP) present in these applications using, primarily, software modifications with minimal hardware support. Exploiting recurrence of events it accelerates these asynchronous programs while saving energy. Having been designed primarily in software it is more flexible and can be deployed on multiple underlying hardware substrates like SMT and CMPs. For a similar set of Web 2.0 applications as above, EER improves performance

by an average of 43% while saving 18% energy.

The techniques presented in this thesis exploit the fact that events are largely independent of each other but not completely so. An execution model and programming language that supports parallel execution could possibly help programmers express parallelism across independent events, diminishing the advantage afforded by architectures like ESP and EER. Despite this, the underlying JavaScript execution model is sequential. However, this is by design and not lack of rigor. Keeping JavaScript sequential helps make it more programmable allowing it to be more accessible to the wide community of web developers, a significant proportion of whom are not likely to be expert programmers.

Moreover, parallel programming with time-shared exclusive access to shared resources can lead to unpredictable execution times of event handlers. This is a huge impediment since user interactions must be responsive which are currently ensured by short event handlers executing sequentially, thus leading to more predictable execution times.

Design spectrum: This thesis has presented an in-depth evaluation of three different designs to accelerate asynchronous programs. However, these are three design points in a spectrum of different designs, some of which this thesis has studied. Others are future research directions that can be pursued based on lessons learnt from this thesis.

In moving from ESP to EER, we dropped the cachelets in favour of reserving a way in the L1 caches for isolation. While this helps reduce the hardware overhead, which was one of the design goals of EER, the hardware changes it requires for speculative pre-execution are arguably more complex than a distinct hardware structure like the cachelets. Also, in case of a 2-way L1 cache, one reserved way for speculative pre-execution will be half the cache leaving too little for the main thread's execution, potentially degrading performance.

To isolate speculative data updates, there is a middle ground between the two designs. One can avoid carving out space from the L1 data cache while still keeping the hardware overhead low by using a significantly smaller data cachelet than the one used in ESP. This can be achieved by caching the data updates of a select few stores, while ignoring others. Different stores in the program can be profiled and the impact of their data updates on subsequent control flow studied. Only data updates of stores that cause the most change in control flow will be cached. This could minimize the impact on accuracy of speculative pre-execution with minimal hardware overhead.

While the I-cachelet was not required for correctness, it helped avoid destructive interference. Without using it or reserving a way in the L1 instruction cache, isolation of the working set of the speculative pre-execution can be achieved through software techniques like page colouring.

Something similar could also be used to isolate speculative data updates. A specific part of the virtual address space could be reserved for the speculative pre-execution, and all address mapping could be handled through a large hash table. While the overhead of this purely software technique could be prohibitive, this can be a feasible solution with smart optimizations.

It is conceivable that one could memoize the events' execution without speculative pre-execution. Theoretically, one could even use the binary instrumentation tool, Pin[34], to record all the instruction and data addresses accessed. However, the overhead of an unoptimized implementation of this technique for interactive applications like websites and mobile apps would be unacceptably high.

EER describes the need to record multiple variants of an event. Conceivably, informa-

tion could be gained through offline profiling where the performance overheads would be more tolerable. This is, however, infeasible in the case of websites' JavaScript execution since the code is compiled at runtime. Also, importantly, the event variants depend on the browsing habits of the specific user.

While there are challenges in building a pure software solution to exploit the insights and techniques presented by this thesis, the benefit is that it is more likely to be adopted since it can run on today's commodity hardware.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Chromium. https://www.chromium.org/Home. 98, 106

[2] Octane benchmark suite. https://developers.google.com/octane/. 31, 70, 106

[3] SPEC CPU 2006. http://www.spec.org/cpu2006/. 19

[4] Sunspider benchmark suite. https://www.webkit.org/perf/sunspider/sunspider.html. 31, 70, 106

[5] Why threads are a bad idea. http://web.stanford.edu/ ouster/cgi-bin/papers/threads.pdf. 1, 48

[6] D. Anderson, F. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967. 43, 72, 81

[7] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 52–61. IEEE, 2001. 82

[8] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database

applications. *ACM Transactions on Computer Systems (TOCS)*, 21(4):412–444, 2003. 39, 42, 45, 46, 115

[9] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 26–37. IEEE, 2001. 51, 75, 81, 116

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. 19

[11] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011. 33, 70, 106

[12] G. Chadha, S. Mahlke, and S. Narayanasamy. Efetch: optimizing instruction fetch for event-driven web applications. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 75–86. ACM, 2014. 81, 82, 115, 116

[13] G. Chadha, S. Mahlke, and S. Narayanasamy. Accelerating asynchronous programs through event sneak peek. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 642–654. ACM, 2015. 107, 116

[14] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society. 81, 116

[15] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*, pages 593–601. IEEE, 1997. 4, 15, 44

[16] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, 1995. 82

[17] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 69–73. ACM, 1991. 82

[18] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th symposium on Operating Systems Design and Implementation*, 2014. 55

[19] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society, 2001. 116

[20] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 14–25, New York, NY, USA, 2001. ACM. 81, 116

[21] J. Doweck. White paper inside intel® core™ microarchitecture and smart memory access. *Intel Corporation*, 2006. 52, 72, 87, 107

[22] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75. ACM, 1997. 51, 75, 81, 116

[23] V. Ekanayake, C. Kelly IV, and R. Manohar. An ultra low-power processor for sensor networks. *ACM SIGARCH Computer Architecture News*, 32(5):27–36, 2004. 118

[24] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011. 4, 14

[25] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 152–162, New York, NY, USA, 2011. ACM. 4, 14, 15, 16, 26, 39, 42, 44, 81, 115

[26] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2008. 14

[27] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 58–69, New York, NY, USA, 1998. ACM. 6, 50, 117

[28] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks. An ultra low power system architecture for sensor network applications. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 208–219. IEEE Computer Society, 2005. 118

[29] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993. 117

[30] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373. IEEE, 1990. 4, 15, 43

[31] A. Kolli, A. Saidi, and T. F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 260–271. ACM, 2013. 39, 42, 45, 115

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009. 43, 72

[33] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in

simultaneous multithreading processors. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 40–51. IEEE, 2001. 44

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the '05 Conference on Programming Language Design and Implementation*, pages 190–200, June 2005. 123

[35] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 18–29. ACM, 2002. 6, 50, 117

[36] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster web browsing using speculative execution. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 9–9. USENIX Association, 2010. 81, 117

[37] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 370–381, Washington, DC, USA, 2005. IEEE Computer Society. 51, 75, 81, 116

[38] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 129–140. IEEE, 2003. 44, 51, 75, 81, 116

[39] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative

to very large instruction windows for out-of-order processors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 129–140. IEEE, 2003. 72

[40] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 20. ACM, 2013. 96

[41] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 165–175. IEEE, 1996. 44

[42] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 371–382. IEEE Computer Society Press, 2002. 4, 15, 43

[43] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 3–3. USENIX Association, 2010. 15, 31, 70, 106

[44] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 16–27. IEEE, 1999. 44

[45] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. *Computers, IEEE Transactions on*, 50(4):338–355, 2001. 44

[46] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35. IEEE Computer Society, 1996. 118

[47] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 37–48. IEEE, 2001. 81, 116

[48] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11(12):7–21, 1978. 4, 15, 43

[49] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM. 6, 50, 117

[50] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 225–236. IEEE, 2005. 44

[51] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch history guided instruction prefetching. In *High-Performance Computer Architecture,*

2001. HPCA. The Seventh International Symposium on, pages 291–300. IEEE, 2001. 44

[52] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM. 6, 50, 117

[53] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000. 44, 81, 116

[54] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.3. *HP Laboratories, Palo Alto, CA*, 2008. 43, 72

[55] 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 *ACM SIGARCH Computer Architecture News*, volume 24, pages 191–202. ACM, 1996. 92

[56] V. Uzelac and A. Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 207–217. IEEE, 2009. 61

[57] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access

ordering. In *Proceedings of the 14th international conference on Supercomputing*, pages 167–175. ACM, 2000. 4, 15, 43

[58] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 137–149. IEEE, 2015. 117

[59] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 13–24. IEEE, 2013. 81, 117

[60] Y. Zhu and V. J. Reddi. Webcore: architectural support for mobileweb browsing. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 541–552. IEEE Press, 2014. 81, 118

[61] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 2–13. IEEE, 2001. 44, 81, 116