EFetch: Optimizing Instruction Fetch for Event-Driven Web Applications

Gaurav Chadha Scott Mahlke Satish Narayanasamy

University of Michigan, Ann Arbor
{gauravc, mahlke, nsatish}@umich.edu

ABSTRACT

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 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 paper, we take one step towards creating a core customized to event-driven applications. We observe 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 find 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. 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.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—cache memories

Keywords

Event-driven web applications; JavaScript; Instruction Prefetching

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628103.

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 are 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 asynchronicity 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 [9]. 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 [10, 11]. 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 [19]. 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 [12, 18, 27, 22], past prefetcher designs have significant limitations when applied to event-driven programs. They either rely on spatial locality [12, 22], address access patterns [27], predictability of branches [8], or require fairly large hardware structures [10].

In this paper, 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 observe 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, our 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. In our design, we prefetch 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.

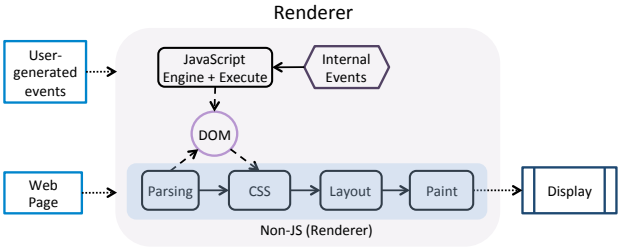


Figure 1: Software components of a renderer process in a browser.

We study 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`). We show that WebCore's prefetcher (EFetch) outperforms commercially implemented next-2-line prefetcher by 17%. It also outperforms a recently proposed prefetcher, PIF [10], while consuming 5.2 times less area.

2. BACKGROUND AND MOTIVATION

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

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 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 1. 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, and then dynamically compiles hot functions into native code.

Web applications are written in JavaScript using an event-driven programming model. Figure 3 shows an example execution of a web application. A looper thread in the renderer process constantly polls an event-queue. It dequeues one event at a time and invokes the necessary JavaScript handler function to process the event. The events may be generated internally or in response to an external input. A critical rule that web developers adhere to is that durations of events are short. This is necessary to ensure that a web application is responsive to the user. 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 Motivation for WebCore

Characteristics of event-driven programs are significantly different from the conventional programs. We studied several microarchitectural characteristics of web applications

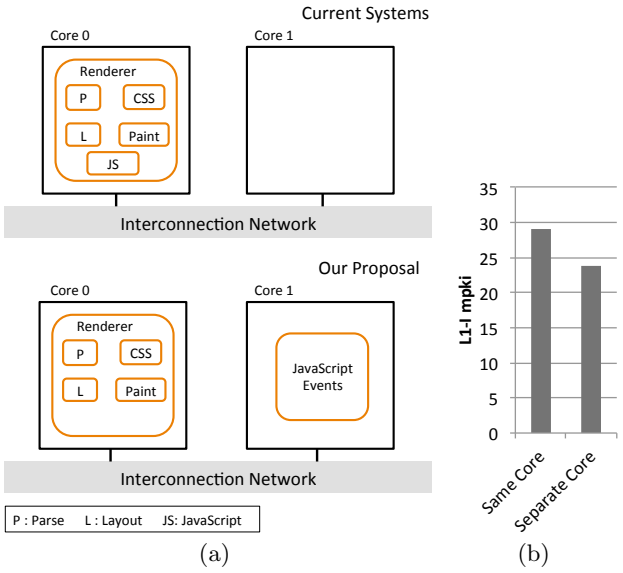


Figure 2: (a) Isolating JavaScript Events, (b) L1-I mpki when executing JS events along with the entire Renderer (Same Core) or Isolated (Separate Core)

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 2(a).

Figure 2(b) 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 4 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 [6] and SPECint 2006 [2] workloads. I-cache mpki for web applications could be as high as 26 (`bing`), where for SPECint 2006 it is about 2.

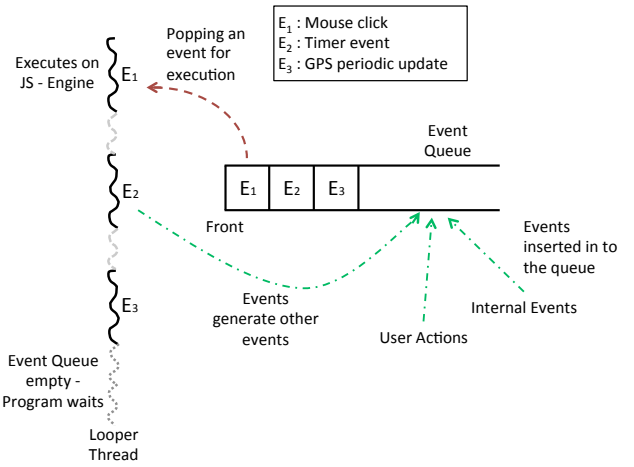


Figure 3: Event-Driven Programming Model

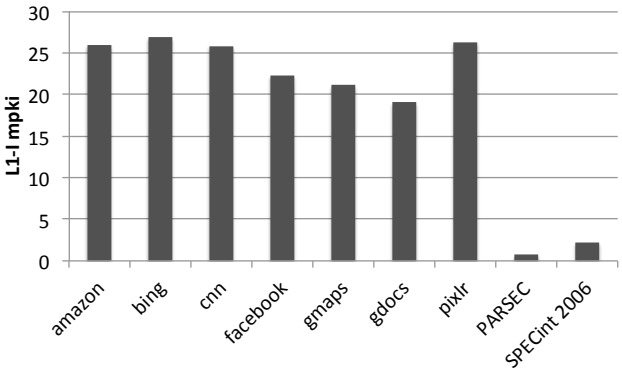


Figure 4: Comparison of L1-I cache mpki

Figure 5 shows the cumulative I-cache L1 miss rates observed due to different cache blocks accessed. As the figure shows, we need over 10,000 cache blocks 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.

3. DESIGN

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

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

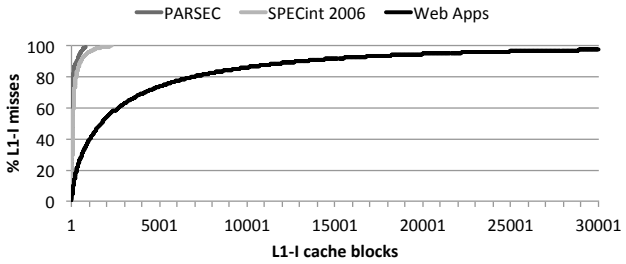


Figure 5: Cumulative Distribution of L1-I cache misses across cache blocks

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 7) 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.

3.2 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 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 7, when f_1 gets called, g_1 is prefetched (step 0 in Figure 8). 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, we prefetch 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.

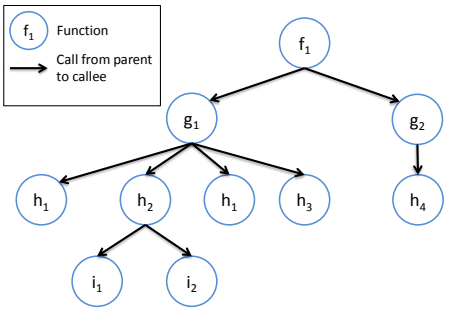


Figure 7: A pre-order traversal of this call graph shows the order in which the functions are called.

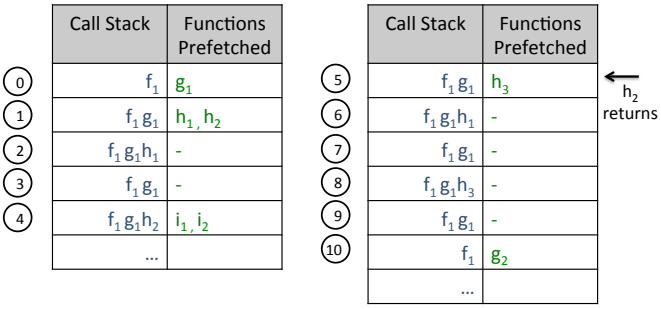


Figure 8: Functions prefetched on different points of program execution

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, we prefetch 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 Predicting Callees and Prefetching Future Accesses

EFetch design architecture has been illustrated in Figure 6. 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 6), one

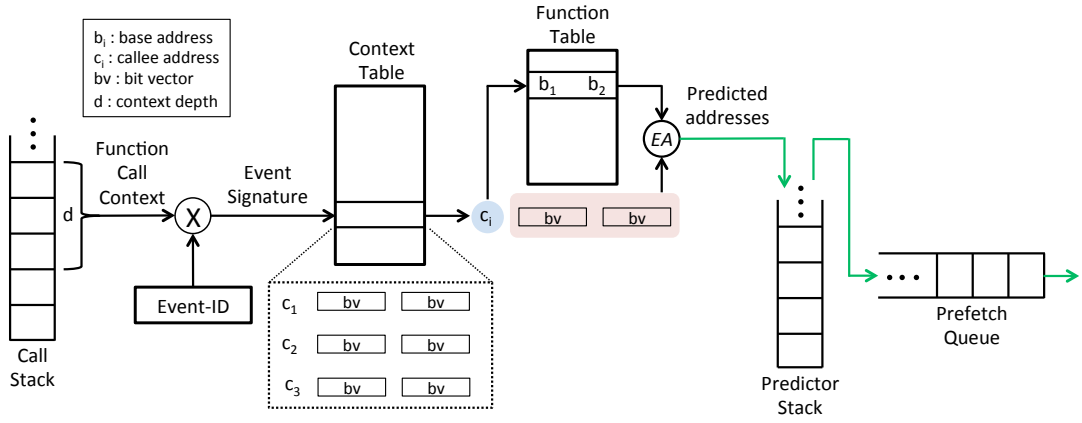


Figure 6: Prefetcher Design Overview.

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 3.4).

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 3.4).

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*.

3.4 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 observe that the L1-I cache blocks accessed in a function are contiguous with a few discontinuities. Therefore, we record pairs of a base address with an associated bit vector representing the nearby blocks, similar to the scheme used in [10].

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 6), 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 6). 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.

3.5 Prefetching Algorithm and Example

In our design we predict 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 9(a)), 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 9(b)), 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 8 shows which function is prefetched relative to program execution. Following the algorithms described in Figures 9(a) and 9(b), Figure 10(a) shows a step-by-step walk-through of the states of the Predictor Stack as an example program executes. At the start of the call graph (Figure 7), 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

// 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 9: Prefetching Design Algorithms

was its second call by its parent - g_1. This is, therefore, not a misprediction. This function has already been prefetched.

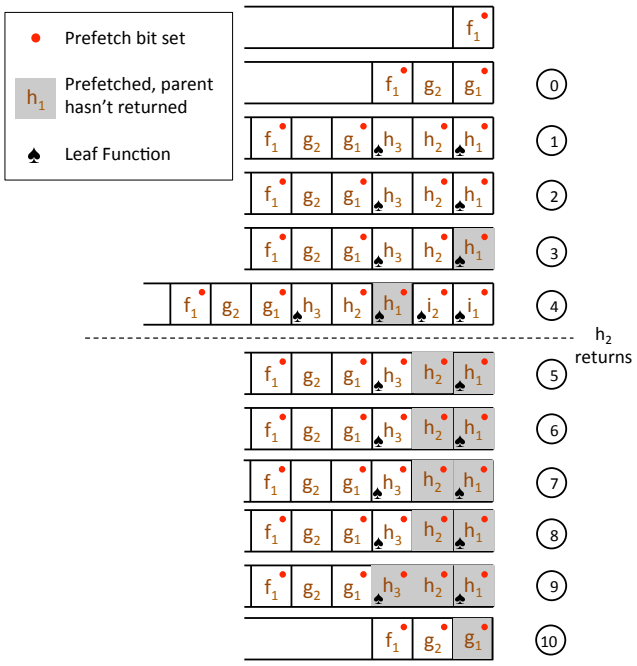
The above is a case where the prefetcher always predicted correctly. Figure 10(b) shows a case where everything proceeds as before until h_3 is called, at which point a misprediction is detected, since the next predicted callee is h_6.

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*.

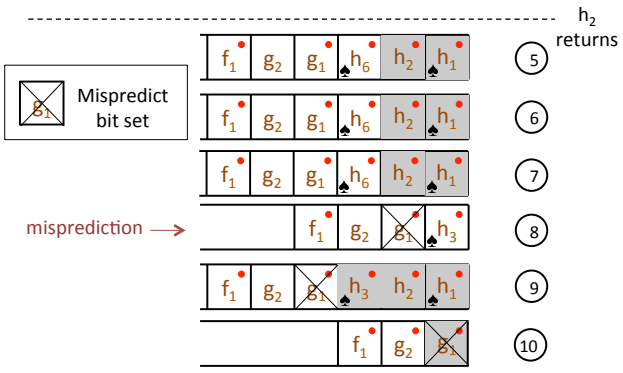
When h_3 returns, the top of the stack (g_1) has a Mispredict bit set, so nothing more is done.

4. METHODOLOGY

In this work we have attempted to evaluate the behavior of event-driven JavaScript (JS) web workloads. This



(a) Predictor Stack



(b) Predictor Stack on a misprediction

Figure 10: Prefetching example

nature of JavaScript execution is not captured by existing JavaScript benchmark suites like Octane [1] and Sunspider [3]. Real-world JavaScript web workloads, as executed in the rendering of popular web pages, bear little resemblance to the benchmark suites, as shown in [19]. 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.

4.1 Web Applications

Figure 11 lists the 7 real web applications that we have used in this study. These websites were chosen since they are both important and popularly visited, and they cut across a 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).

Web Site	Actions performed	#events executed	# JS-Execute instructions
amazon (amazon.com)	Search for a pair of headphones, click on one result, go to a related item	7,787	432,875,253
bing (bing.com)	Search for the term "Roger Federer", go to new results	4,858	258,961,747
cnn (cnn.com)	Click on the headline, go to world news	13,409	1,230,395,548
fb (facebook.com)	Visit own homepage, go to communities, go to pictures	9,305	2,165,554,555
gmaps (maps.google.com)	Search for driving directions between two addresses, get public transit directions, get biking directions	7,298	2,722,470,912
gdocs (docs.google.com)	Open a spreadsheet, insert data, add 5 values	1,714	808,941,337
pixlr (pixlr.com/editor)	Add various filters to an image uploaded from the computer	465	26,424,174

Figure 11: 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 GB DRAM, 101 cycle access latency, 12.8 GB/s bandwidth
Branch Predictor	Pentium M branch predictor 15 cycle mis-predict penalty 2k-entry Global Predictor, 256-entry iBTB 2k-entry BTB, 256-entry Loop Branch Predictor
Interconnect	Bus
Energy modeling	V_{dd} = 1.2 V, 45 nm

Figure 12: Details of the architecture simulated

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 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.

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 us 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 Figure 11. In order to create a workload that is repeatable, we captured the instruction trace of the browser [1], using the trace-recording component

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

of SniperSim [7]. 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 Figure 12.

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.

5. RESULTS

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

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. We, therefore, keep 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.

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 are trying to predict the very next function to be called, staying one function ahead of program execution.

In Figure 14, 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.

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

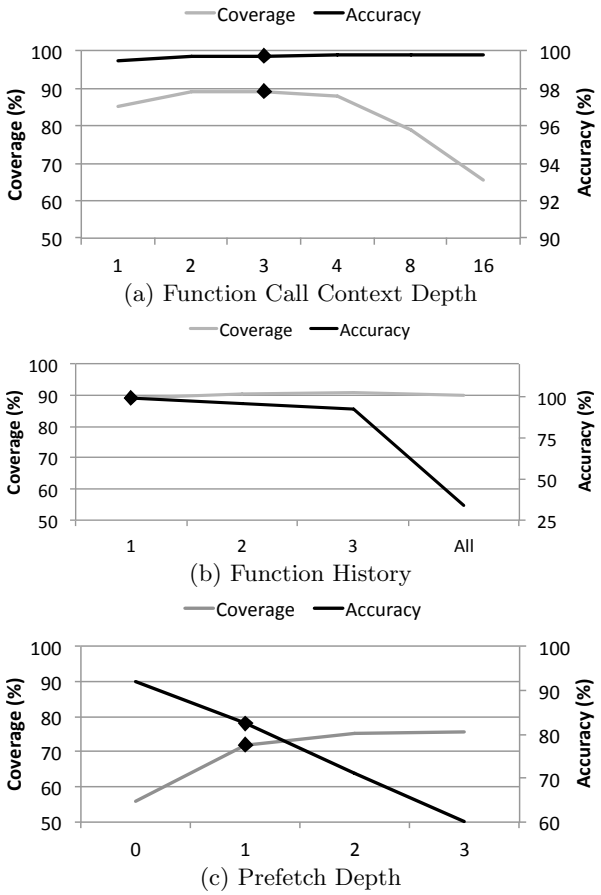


Figure 13: Variation in *Coverage* and *Accuracy* with varying *Function Call Context Depth*, *Function History* and *Prefetch Depth*. The diamond marks the final design choice

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 13(b), 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.

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 cor-

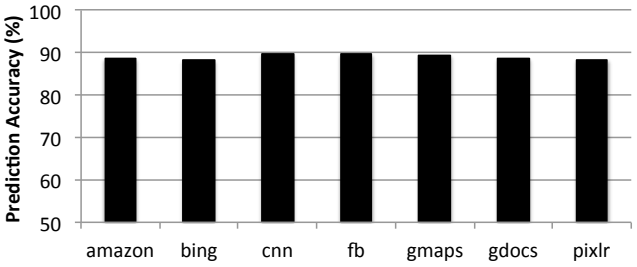


Figure 14: Callee Set Prediction Accuracy

nerstone 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 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 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 13(a) validates the above hypotheses. Coverage reduces with increasing context depth without any noticeable change in accuracy.

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 13(c) 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.

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

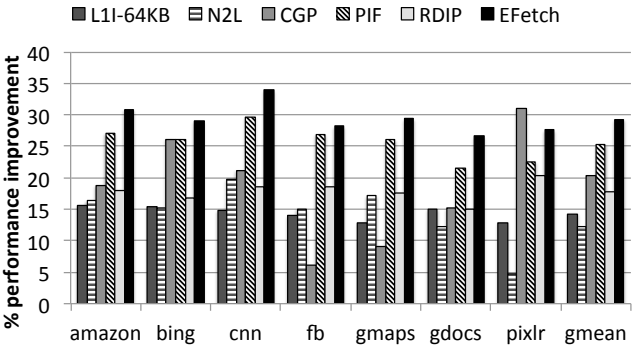


Figure 16: Performance achieved compared to No Prefetching (NP) as the baseline

callee set and function body. A diamond marks our final design parameters in the earlier plots.

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 [5].
- *PIF* is our implementation of the Proactive Instruction Fetch [10].
- *RDIP* is our implementation of Return-Address-Stack directed instruction prefetching [13].
- *EFetch* is our design.

In Figure 15, 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

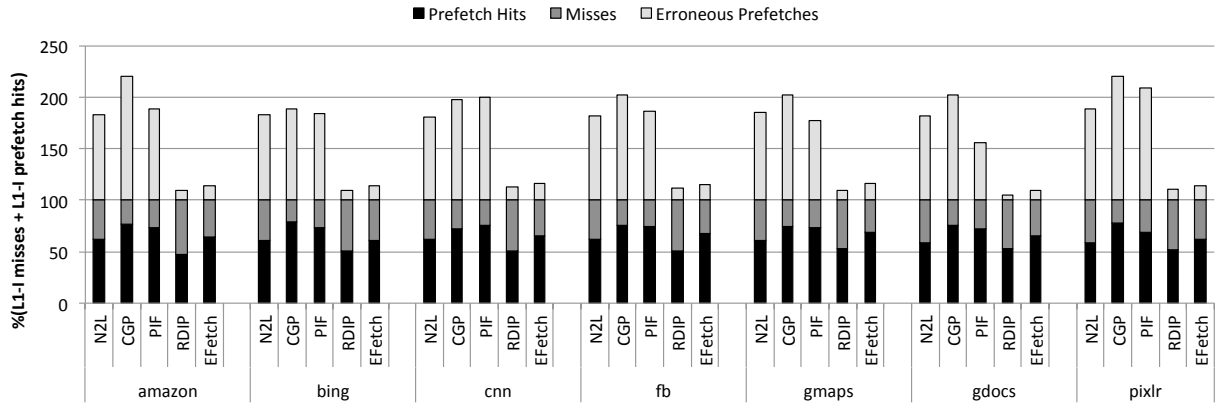


Figure 15: Each bar is divided in to three segments: *Prefetch Hits* are misses removed by prefetching. *Misses* are the remaining L1-I cache misses. *Erroneous Prefetches* are prefetches made which were never used (until their eviction from L1-I cache).

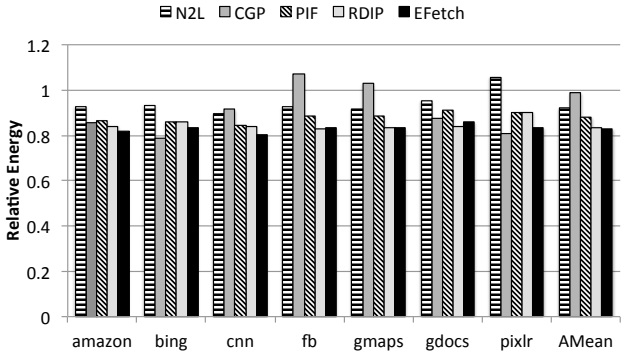


Figure 17: Energy expended compared to No Prefetching (NP) as the baseline

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

Figure 16 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%.

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.

Design	Structure	Size (KB)	Access Energy (pJ)	Static Power (mW)
CGP [5]	Call Graph History Cache	32	44.8	28.6
PIF [10]	History Buffer	136	39.3	119.9
	Index Table	68	25	62.2
RDIP [13]	Miss Table	63	32.6	49
EFetch	Callee Predictor Table	39	21.5	33.9

Table 1: Energy and power estimates used for hardware structures

5.4 Energy

We evaluated the energy expended by the different instruction prefetcher designs using McPat 0.8 [14]. CACTI 5.3 [26] was used to determine the per-access energy and static power of the additional hardware structures as shown in Table 1.[2] Figure 17 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.

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 sequen-

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

tial instruction fetch [4]. This initial idea was expanded upon and next-N-line and instruction stream prefetchers were proposed [12, 18, 27, 22] 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 heavy code, several branch predictor based prefetchers [8, 20, 21, 24] have been proposed. Run-ahead execution [16], wrong-path instruction prefetching [17] and using an idle thread [15] or speculative threads [25, 28] can be used to generate future instruction accesses. Ferdman et. al. [10] 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 [23] 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) [24] 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 [10] 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, we 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 [13], 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 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 [5]. 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 [5], 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.

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 paper 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.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and feedback. This research was supported by National Science Foundation grants CAREER-1149773 and SHF-1217917 and funding from Intel Corporation.

9. REFERENCES

[1] Octane benchmark suite. https://developers.google.com/octane/.

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

[3] Sunspider benchmark suite. https://www.webkit.org/perf/sunspider/sunspider.html.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.