

Abstract

ROGERS, BRIAN MICHAEL

Memory Predecryption: Hiding the Latency Overhead of Memory Encryption.

(Under the direction of Dr. Yan Solihin)

Security has emerged as an important area in the field of computer research today. With the emergence of hardware-based attacks, research has been done not only on software solutions to security, but also on providing security with the help of architectural support. More specifically, hardware encryption and authentication of off-chip memory have recently been studied as ways to ensure that malicious agents cannot see data in its plaintext form or tamper with data in an undetected manner during an application's execution. When used in combination, encryption and authentication can help to provide a secure processing environment.

While various techniques have been proposed for performing memory encryption in a secure processor, current schemes suffer from extra performance and storage overheads. This paper presents *predecryption* as a method of providing this encryption with less overhead. This is accomplished by using well-known prefetching techniques to retrieve data from memory and perform decryption before it is needed by the processor on latency-

critical read operations. Our results, tested mostly on SPEC 2000 and NAS benchmarks, show that using this predecryption scheme can actually result in no increase in execution time over a system with no encryption, despite an extra 128 cycle decryption latency per memory block access.

**MEMORY PREDECRYPTION: HIDING THE LATENCY
OVERHEAD OF MEMORY ENCRYPTION**

by
BRIAN MICHAEL ROGERS

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh
2005

APPROVED BY:

Dr. Gregory T. Byrd

Dr. Jun Xu

Dr. Yan Solihin, Chair of Advisory Committee

BIOGRAPHY

Brian Rogers was born on April 28, 1981, in Greensboro, NC. In May 2003, he received his Bachelors degree in Computer Science, along with a minor in Business Administration from the University of North Carolina at Chapel Hill.

He enrolled at North Carolina State University in August 2003, to begin work on his Masters degree in Computer Engineering. He joined Dr. Yan Solihin's research group in May 2004. Since then, he has decided to continue his studies at North Carolina State University after the completion of the Masters degree to pursue a doctorate in Computer Engineering.

ACKNOWLEDGEMENTS

First of all, I would like to thank my parents, Gary and Carol Rogers, and my brother Scott. Their endless support and guidance both in life and in my academic career have been such a blessing to me. They have made so many things so easy for me, and I could never fully express what they have meant to me. Without their help, none of this would have been possible for me.

I would like to thank Dr. Yan Solihin for giving me the opportunity to work under his guidance and support. He has made me feel like a part of his research group from day one, and I have learned countless things in the past year while doing research, including much about computer architecture, how to conduct research effectively and efficiently, how to write research papers, how to more effectively give oral presentations, and how to manage time wisely. The concern and time that Dr. Solihin puts into preparing his students for their future is always evident, and I want to thank him for that as well.

I would like to thank Mazen Kharbutli for being my mentor when I joined the research group, and giving me lots of valuable knowledge and understanding into the SESC simulator used by our group. I would also like to thank Seongbeom Kim, Rithin Shetty, Dhruba Chandra, Fei Guo, and Radha Venkatagiri who were never too busy to offer advice or suggestions to problems that I encountered during the past year.

I would also like to thank Dr. Jun Xu and Dr. Gregory T. Byrd for agreeing to be on my advising committee and evaluate the work of my Masters Thesis.

Finally, I would like to thank Brandon Boggs, Jared Wilson, and Jimmy Joyner for the best roommate experience you can ask for during my Masters study. All the good times we had together have no doubt helped me during the tough, stressful times. Thank you for all of your advice and great friendship. And I would like to thank Lindsay Bowers for being such a special part of my life over the past year, and for helping to keep me focused and motivated on my work.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
1. Introduction	1
1.1 Current Approaches	1
1.2 Our Approach	3
1.3 Paper Organization	4
2. Predecryption Mechanism	5
2.1 Qualitative Comparison with OTP	7
3. Prefetching Algorithms	10
3.1 Stream Prefetcher	10
3.2 Correlation Prefetcher	12
4. Evaluation Setup	15
5. Evaluation	18
6. Conclusions	26
References	27

LIST OF TABLES

Table 1: Qualitative comparison between our predecryption approach and OTP-approximation approach.	7
Table 2: The applications used in our evaluation.	15
Table 3: Parameters and configurations of the simulated architecture.	16

LIST OF FIGURES

Figure 1: Predecryption mechanism for a processor with two levels of on-chip caches.	6
Figure 2: Structure of the Correlation Table, and an example of its operation with a sample miss stream.	13
Figure 3: SPEC2K app. performance with different predecryption schemes	19
Figure 4: Non-SPEC app. performance with different predecryption schemes	19
Figure 5: Average performance of all apps. with different predecryption schemes	19
Figure 6: Predecryption benefit over standard prefetching for SPEC2K apps.	21
Figure 7: Predecryption benefit over standard prefetching for non-SPEC2K apps. . . .	21
Figure 8: Average predecryption benefit over standard prefetching for all apps.	21
Figure 9: Prediction performance of SPEC2K apps. for different predecryption schemes.	22
Figure 10: Prediction performance of non-SPEC2K apps. for different predecryption schemes.	22
Figure 11: Average prediction performance of all apps. for different predecryption schemes.	22

1. Introduction

Increasingly numerous and sophisticated security threats are creating a growing need for processing environments resistant to software piracy and security attacks. While remote software-only attacks are receiving much of the attention in this area, hardware attacks are also possible and feasible. For example, on the X-Box game console a bus or DRAM override can be accomplished by inserting an inexpensive chip on the bus, allowing unauthorized playing of games [7]. Such violations of the Digital Rights Management (DRM) policy can be very costly - software piracy has cost the software industry billions of dollars per year in recent years due to unauthorized copying of restricted software [9]. One promising solution for hardware-based attacks is to encrypt data as it leaves the processor chip and decrypt it when it is brought back on-chip.

1.1. Current Approaches

Recent studies in computer architecture have proposed a model referred to as the execute-only memory (XOM), as a way to support copy and tamper resistant software communication [5, 13, 14]. In XOM, application programs and their data are stored encrypted in the main memory. Unfortunately, as observed by other researchers, the performance overhead of using memory encryption in this way is very high, slowing down application programs by up to 35% even when using a very simple encryption algorithm that takes only 48 cycles to decrypt a cache line [21]. This is because each memory access suffers from decryption latency on its critical path. Furthermore, this decryption latency may actually

take up to hundreds of cycles using the popular AES algorithm.

To lower performance overheads, recently proposed schemes rely on approximating one-time pad (OTP) encryption [20, 21] to allow overlap between a memory access and decryption. In these schemes, memory blocks are not directly encrypted using a strong key-based scheme. Instead, the key is used to encrypt a “unique identifier” (pad) of the block, obtained from the block’s address and a sequence number. The resulting pad for the block is then XORed with the plaintext of the block to produce ciphertext. When the block is fetched from memory, it is decrypted by recomputing its pad and XORing it with the encrypted block. The fetch of an encrypted block from memory and the pad computation for that block can be overlapped, hiding much of the latency of encryption/decryption. However, this approach has a significant drawback: the pad must be different each time a block is encrypted [20, 21]. If a pad is reused and the attacker knows, discovers, or guesses the plaintext of one data block, it is a simple matter to recover the pad (by XORing the known plaintext and its ciphertext) and use it to decrypt all other blocks that used the same pad. In memory systems, many locations have known or easily guessable values (e.g. many are zeroes), so using the same pad more than once is very risky. Because the address of a data block remains constant, the uniqueness of the pad for different versions of that block relies on the sequence number used to generate the pad. Maintaining such sequence numbers is a significant problem. The sequence number is kept on a per-block basis, so the OTP-like scheme must record the current sequence number for each block in memory. Pad generation cannot begin until the block’s sequence number is fetched, so most of the

latency-hiding opportunity is lost unless sequence numbers are kept in fast and relatively large on-chip storage. On-chip storage is still not sufficient for sequence numbers of *all* blocks in memory, so storage of sequence numbers also imposes a significant memory space overhead. Finally, when sequence numbers wrap around, the encryption key must be changed to avoid using the same pads again. This requires re-encryption of the entire memory, which is a significant overhead. The frequency of wrap-arounds can be reduced by using large sequence numbers, but then even more space is needed to store them.

1.2. Our Approach

To avoid the complexities and disadvantages of OTP-approximating schemes, we evaluate the feasibility of using well-known latency-hiding prefetching techniques to minimize the overheads of memory encryption. We augment the processor with a prefetching engine that predicts future cache misses, prefetches, and decrypts them ahead of the processor’s requests. We call this scheme *predecryption*. The prefetcher that we use includes stream buffers [3, 6, 8, 11, 15, 17] and a correlation prefetcher [1, 2, 10, 12, 17, 18]. Compared to XOM and OTP, this approach has several benefits. First, prefetching engines are already present in real systems [6, 8], so little extra hardware is needed, other than tuning it to also hide the decryption latency. Furthermore, more advanced predictors will not only hide decryption latency better, they may also improve performance by also hiding memory latency. Second, predecryption allows well-known encryption schemes, such as AES, to be directly used to encrypt data in memory. In contrast, OTP-like schemes hide decryption

latency by modifying the encryption scheme. Third, unlike OTP-like schemes, predecryption does not use sequence numbers, so no memory space is needed for them and there is no need to periodically re-encrypt the entire memory when a sequence number wraps around. Finally, when the data needs to be communicated to other devices, such as other processors in a multiprocessor system, the key needs to be passed to these devices only once. In OTP-like schemes, the sequence number needed to generate the pad for a block must be communicated between devices together with the encrypted block itself.

1.3. Paper Organization

The paper is organized as follows: Section 2 discusses the predecryption algorithm used and compares it with OTP, Section 3 describes the stream prefetcher and correlation prefetcher in more detail, Section 4 details the evaluation setup, Section 5 presents and discusses the evaluation results, and Section 6 summarizes our findings and conclusions.

2. Predecryption Mechanism

Figure 1 shows the mechanism for predecryption. When an L2 cache line is replaced or flushed, it is encrypted before it is written back to the memory (Step 1a). A write-back is typically not a latency-critical operation; therefore we do not attempt to hide the encryption latency. Instead, we mainly focus on hiding the decryption latency, which affects time-critical fetches from memory into the L2 cache upon an L2 cache miss. When an L2 cache miss occurs, instead of requesting the line from memory, we first check the predecryption buffer for a match (Step 1b). If the missed line is found in the predecryption buffer, it is moved to the L2 cache. Otherwise, the miss is forwarded to the memory controller for fetching and to the prefetcher (Step 2). The prefetcher’s stream buffers use the miss to identify streams and predecrypt them. If the miss is not identified as a part of a stream, the miss is forwarded to the correlation prefetcher to record the address in its table and make its prediction on future misses. Thus, the correlation prefetcher only sees addresses that are not sequential. In this way, the correlation table can be smaller because it is only used for “difficult” misses that can not be handled by stream buffers. A similar optimization has been used in past studies on prefetching [17, 18].

When the prefetcher observes a miss, its stream buffers or its correlation prefetcher predicts future miss(es) and issues a predecryption request to the memory controller (Step 3). When the main memory replies with data (Step 4), if it is a reply to a read/write miss, it is decrypted and inserted into the L2 cache (Step 5a). If it is a reply to a predecryption request, the data is decrypted and stored in the predecryption buffer (Step 5b).

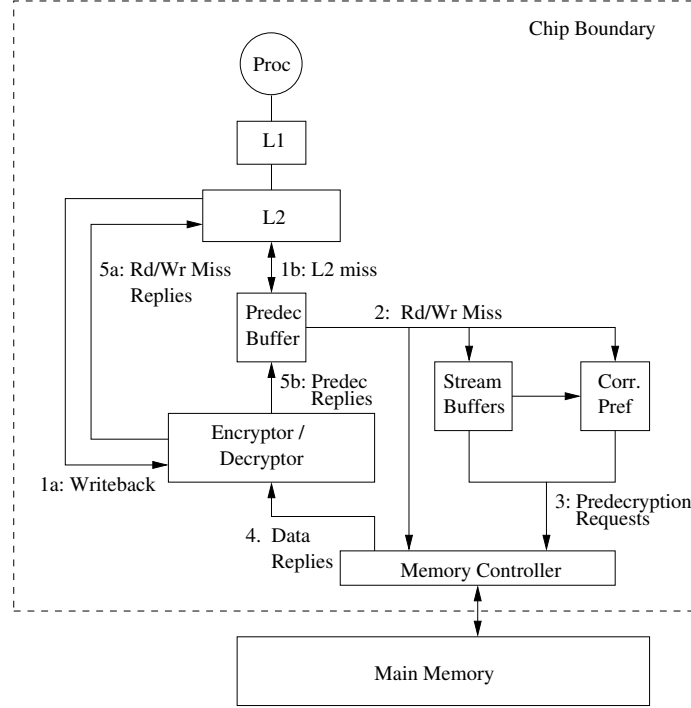


Figure 1: Predecryption mechanism for a processor with two levels of on-chip caches.

As long as the predecryption mechanism is able to predict far enough ahead, both the decryption latency and the memory latency can be hidden. Therefore, we tune the prefetcher to predict future misses far into the future. To achieve that, we use deep stream buffers, with eight entries per buffer, and other standard features such as a multiple-stream detection capability, double Δ scheme, and a heuristic to avoid storing overlapping streams [4]. For correlation prefetching, we use a replicated table organization that prefetches several levels of successors for the current miss address [18]. A more detailed description of the prefetchers appears in the next section.

2.1. Qualitative Comparison with OTP

Table 1 qualitatively compares the properties of our predecryption scheme with encryption based on One-Time Pad (OTP) approximation, which was previously proposed [20, 21]. For a cache miss, in the best case scenario, OTP finds the needed sequence number on chip in the *sequence number cache* (SNC), and can overlap pad generation latency with memory latency. The best case for predecryption is when the requested line has already been fetched and decrypted into the predecryption buffer, in which case predecryption completely hides decryption and memory latencies. The worst-case latency for OTP is when the sequence number is not found in the SNC and needs to be fetched before decryption can be performed, in which case the sequence number must be retrieved from memory before pad generation can begin. Therefore, the overall latency of the entire operation is the sum of memory and pad generation (encryption) latency. The worst-case latency for predecryption is the same, because a cache line must be fetched and then decrypted before it can be used by the processor.

Table 1: Qualitative comparison between our predecryption approach and OTP-approximation approach.

Aspect	OTP-Approximation Approach	Predecryption Approach
Best-case latency	Cache <i>miss</i> latency	Predecryption buffer <i>hit</i> latency
Worst-case latency	Cache miss + decryption latency	Cache miss + decryption latency
On-chip storage overhead	Sequence Number Cache	Stream buffers + correlation table
Off-chip storage overhead	Sequence number storage	None
Other overheads	Re-encrypt entire memory on seq. num. wraparound	None
Communication with other procs	Transfer sequence number on each instance + key once	Transfer key once

In terms of storage, OTP stores a sequence number for each block in memory, either in the SNC on-chip, or off-chip. Predecryption requires prediction information on-chip. If only stream buffers are used, they occupy little storage space, in our case 1KB to store information from multiple streams, 64B to store the addresses of the previous misses, and 16KB for their predecryption buffer. However, to perform well a correlation table may need to be very large, such as 1 MB used in past studies [18, 1, 2, 10, 12]. Reducing the size of the correlation table without significantly affecting its effectiveness is a topic for future work.

OTP also complicates communication with other processors in a multiprocessor environment and with other devices. When an off-chip device accesses a datum, it needs the encryption key and the sequence number. The sequence number is different for each line and must be fetched from memory or, if the number is still in the SNC, from the processor that last updated the line. This could be a significant overhead, especially for parallel or I/O intensive programs. In contrast, with predecryption external devices only need the key, which can be supplied to them once. Therefore, predecryption has virtually no additional run-time overhead for parallel and I/O intensive programs.

Finally, it should be noted that it may be possible to combine our predecryption scheme with an OTP scheme to take advantage of the benefits of each, and we plan to look into this as future work. More specifically, by prefetching both data and sequence numbers the best case latency can still be just a fetch from our predecryption buffer. In addition, when we do not hit in the predecryption buffer we can still find the sequence

number on-chip, and thus overlap the data fetch with the decryption latency since we are using OTP-based encryption. In other words, some of the L2 cache misses that suffer the worst case latency in our predecryption scheme can instead suffer from the best case latency of the OTP-based scheme. A small percentage of L2 misses will still suffer from the worst case latency, which is the same for both schemes.

Overall, in terms of best-case latency, on-chip and off-chip storage overheads, and complexity, predecryption is a very promising alternative to OTP-approximation encryption schemes.

3. Prefetching Algorithms

This section describes the stream and correlation prefetch units in greater detail. As mentioned, when both prefetchers are utilized, the correlation prefetcher only sees misses that are not part of sequential streams so that it can use its storage space to keep track of only the more difficult miss patterns.

3.1. Stream Prefetcher

Stream prefetching [3, 6, 8, 11, 15, 17] is used to capture and prefetch data for sequential access streams in an application. Stream buffers [11] are FIFO queues that are used to identify and prefetch for only increasing, strictly sequential access streams. The prefetched data is held in the stream buffers themselves, and then moved into the cache when a cache miss address matches the head of a stream buffer. In [15] this mechanism is extended in several ways. They introduce a filtering scheme where a stream is only identified and tracked when two sequential access are observed in a series of misses. They also introduce a method to track not only increasing sequential streams, but also increasing or decreasing, non-unit stride patterns. Our scheme is somewhat of a combination of these two in that it only prefetches sequential stream accesses, but the streams can be either increasing or decreasing. Our scheme uses the double Δ enhancement to be even more selective in prefetching, and it stores its prefetched data in a separate prefetch buffer, not in the stream tracking structures themselves. The specifics of our implementation are given below.

The stream prefetcher utilizes two structures in addition to the prefetch buffer which holds prefetched, decrypted cache blocks. One structure is a FIFO queue which stores the addresses of the last N misses. The other structure is used to identify a limited number of current streams that have been detected by the stream prefetcher.

When an L2 cache miss occurs, the stream prefetcher is accessed. If the block is found in the prefetch buffer, then it is removed from the prefetch buffer and placed into the L2 cache. Then the structure that identifies the current streams is checked to see if it is still tracking the stream where the prefetch buffer hit came from. If so, then requests are generated for the next sequential data blocks from the stream, and the stream information is updated to reflect the current state. It should be noted that in all cases, predecryption requests are only generated if the block is not already stored in the prefetch buffer or in the L2 cache. If the block is not found in the prefetch buffer, then the following steps are taken. First, the block address of the miss is compared to each address in the queue of the N previous misses. Since a double Δ scheme is used, the miss is determined to be part of a sequential stream if the previous miss queue contains block addresses that are plus one and plus two or minus one and minus two from the block address of the current miss. The double Δ scheme helps to limit the number of false positive stream identifications, and thus limit the number of useless predecryption requests that are generated. After checking to see if the current miss is part of a new stream, its block address is pushed into the previous misses queue. If a stream has been detected, then an entry is allocated in the stream identifier structure, possibly replacing an old entry, and requests are generated for the next

sequential data blocks.

3.2. Correlation Prefetcher

Correlation prefetching [1, 2, 10, 12, 17, 18] is used to capture and prefetch seemingly erratic, but repeating data access patterns in an application. Our correlation prefetching algorithm is based on the *Replicated* correlation table structure described in [18]. The table structure and an example of its operation on a sequence of L2 cache misses is shown in Figure 2.

The table has four parameters used that describe its setup: *NumRows*, *Assoc*, *NumSucc*, and *NumLevels*. *NumRows* and *Assoc* describe the number of rows in the table and the associativity of the table respectively. Each table row stores the tag of an address and *NumLevels* columns with each column holding *NumSucc* addresses of successor misses. There are also *NumLevels* pointers that each point to a particular row. These pointers identify location of the previous miss, the second previous miss, and so on up to *NumLevels*. Two operations are performed on the table each time an L2 cache miss address reaches the correlation prefetcher. The first operation, determining the addresses to prefetch, simply involves indexing the table with the address and searching for a matching tag. If a match is found, all the successor addresses found in that table row are sent for predecryption as long as the address does not already reside in either the L2 cache or the prefetch buffer. The second operation, updating the correlation table, is performed after all the prefetch addresses have been generated. If the miss address does not occupy a row in the table, one

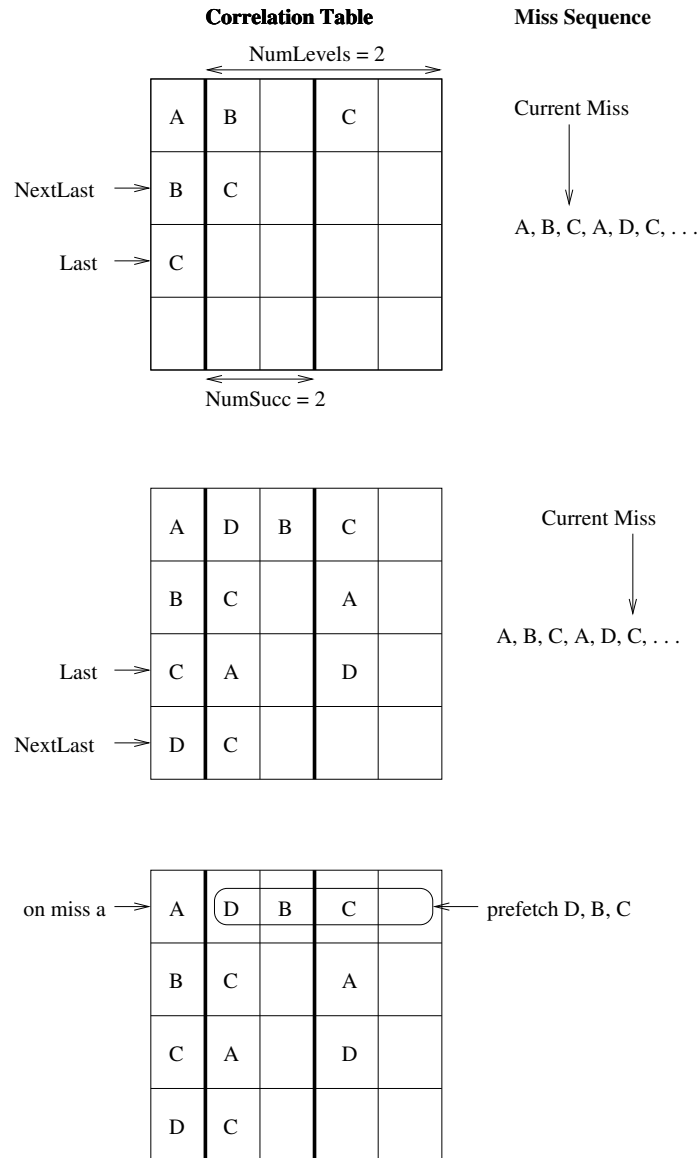


Figure 2: Structure of the Correlation Table, and an example of its operation with a sample miss stream. This example is taken from [18].

is allocated, possibly overwriting an existing row. Then the table rows pointed to by the *NumLevels* pointers, are accessed one by one, and the current miss address is stored in one of the *NumSucc* entries at the corresponding level. The successors within each level are stored in MRU order, so if the level was already full then the LRU successor is pushed out and the current miss address is inserted. After this has been performed for each level, the previous miss *NumLevels* pointers are updated to point to the correct table rows.

4. Evaluation Setup

To evaluate our approach, we use 18 applications, mostly from the Spec2000 [19] and NAS benchmark suites. Table 2 lists these applications and their characteristics. The selected applications show varying sizes and L2 cache miss rates. The evaluation is done using the SESC simulator from the Univeristy of Illinois. The SESC simulator performs detailed execution-driven simulation of an out-of-order, superscalar processor. The relevant simulation parameters are shown in Table 3, together with descriptions of specific configurations used in our experimental evaluation.

Table 2: The applications used in our evaluation.

Benchmark	Source	Input Set	L2 Cache Global Miss Rate	L2 Cache Local Miss Rate	Dynamic Instructions
applu	Spec2K	train	2.96%	81.71%	15,559 M
bt	NAS	class A	0.09%	4.37%	560 M
bzip2	Spec2K	train	0.15%	8.02%	6,696 M
cg	NAS	class S	1.88%	18.90%	289 M
equake	Spec2K	train	2.87%	78.90%	25,120 M
euler	NASA	euler.in	0.51%	2.4%	2,793 M
ft	NAS	class S	1.91%	15.86%	1,567 M
gap	Spec2K	train	2.72%	55.42%	6,733 M
irr	PDE solver	10K nodes and 1M edges	1.52%	3.41%	963 M
is	NAS	class A	5.45%	35.55%	3,373 M
lu	NAS	class A	3.49%	55.19%	2,844 M
mcf	Spec2K	train	7.16%	26.39%	6,398 M
mgrid	Spec2K	64 × 64 × 64 grid, 3 iters	1.41%	55.05%	28,603 M
moldyn	Molecular dyn. code	moldyn.in	0.16%	2.63%	4,954 M
mst	Olden	1024 nodes	1.36%	37.34%	517 M
parser	Spec2K	train	0.39%	9.77%	7,811 M
sp	NAS	class A	2.63%	46.01%	2,825 M
swim	Spec2K	train	7.05%	49.98%	8,380 M

Table 3: Parameters and configurations of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*. We assume an AES encryption/decryption algorithm is used and can be performed in 128 cycles. This is roughly in line with 14-cycle latency on a specialized 154 MHz AES processor reported in [16].

PARAMETERS

Processor	4 GHz, 6-way out-of-order issue Int, fp, ld/st FUs: 4, 3, 4 Branch penalty: 17 cycles ROB size: 248
Memory	L1-Inst: 16 KB, 2 way, 32-B line, WB, RT: 2 cycles, LRU, Outstanding ld+st misses: 16+16. L1-Data: 16 KB, 2 way, 64-B line, WB, RT: 3 cycles, LRU, Outstanding ld+st misses: 24+24. L2-Unif. (shared/per proc): 1 MB, 8-way, 64-B line, RT: 16 cycles, LRU, Outstanding ld+st: 24+24. Predecryption Buffer: 4-way, LRU, 64-B line, 16KB for Sbuff and 32KB for CP (48KB in Sbuff+CP) Memory bus: 1 GHz, 4-Byte wide, split-transaction RT memory latency: 275 cycles
Stream Buffers	Maximum 16 streams, 8 entries/buffer
Correlation Table	Replicated organization [18], 64K entries, 2-way, 2 levels, 2 successors/level, 8-cycle access, 647KB total
Encr./Decrypt	128 cycles for each cache line

CONFIGURATIONS

NoEnc	No encryption/decryption delay and no predecryption
Enc	Encryption/decryption delay and no predecryption
Sbuff	No encryption/decryption delay and prefetching with stream buffers
Enc+Sbuff	Encryption/decryption delay and predecryption with stream buffers
CP	No encryption/decryption delay and prefetching with correlation table
Enc+CP	Encryption/decryption delay and predecryption with correlation table
CP+Sbuff	No encryption/decryption delay and prefetching with both stream buffers and correlation table
Enc+CP+Sbuff	Encryption/decryption delay and predecryption with both stream buffers and correlation table (SPEC2K benchmarks use delta scheme for the correlation table with this configuration)

The stream prefetch unit has the following parameters in our evaluation. The miss window is size 16, so our previous misses queue holds the previous 16 miss addresses. The stream prefetcher can also keep track of stream information for a maximum of 16 streams

at one time. Finally, each time a miss is detected as part of a stream, the prefetcher attempts to generate predecryption requests for the next 8 sequential data blocks, as long as they are not already stored in the predecryption buffer or the L2 cache.

The correlation table has 64K entries. Each entry contains a tag of 11 bits, and multiple successors, where each successor is stored as an offset to the current miss address, and is encoded with 17 bits. Therefore, the total table size is $(64K \times (11 + 4 \times 17))/8 = 647\text{Kbytes}$. The table is also 2-way associative, with 2 levels per entry, and 2 successors per level.

5. Evaluation

Figure 3 plots the execution time for each SPEC2K benchmark, Figure 4 plots the execution time for each non-SPEC2K benchmark, and the average of all the benchmarks is shown in Figure 5, when no encryption is applied (*NoEnc*), when encryption is applied (*Enc*), and when various predecryption schemes are applied (*Enc+Sbuff*, *Enc+CP*, and *Enc+CP+Sbuff*). All bars are normalized to the *NoEnc* case. Figure 5 shows that on average, adding a 128 cycle encryption/decryption delay to all memory accesses increases the execution time by 21%. Predecryption with stream buffers (*Enc+Sbuff*) reduces this overhead to a little over 1%, whereas a correlation predictor alone (*Enc+CP*) reduces this overhead to 16%. A combination of both predecryption schemes (*Enc+CP+Sbuff*) results in the same execution time as in the *NoEnc* configuration. This result shows that, as explained in Section 1, a predecryptor eliminates some of the performance lost to memory encryption. However, these figures do not indicate what part of the miss latency is hidden by predecryption: memory latency alone, or both memory and decryption latency. Figures 6, 7, and 8 help to answer this question.

Figures 6, 7, and 8 show the execution time reduction due to prefetching/predecryption in a system with or without memory encryption for the SPEC2K benchmarks, non-SPEC benchmarks, and the average over all benchmarks respectively. In each figure, the first pair of bars in each group shows the relative reduction in execution time due to prefetching in a system without memory encryption. The second pair of bars shows the relative reduction in execution time due to predecryption in a system with memory en-

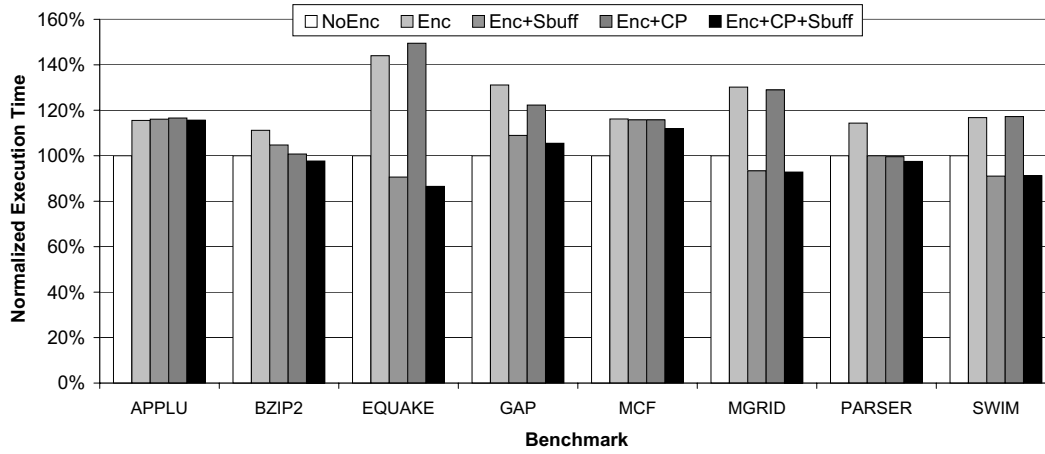


Figure 3: SPEC2K app. performance with different predecryption schemes.

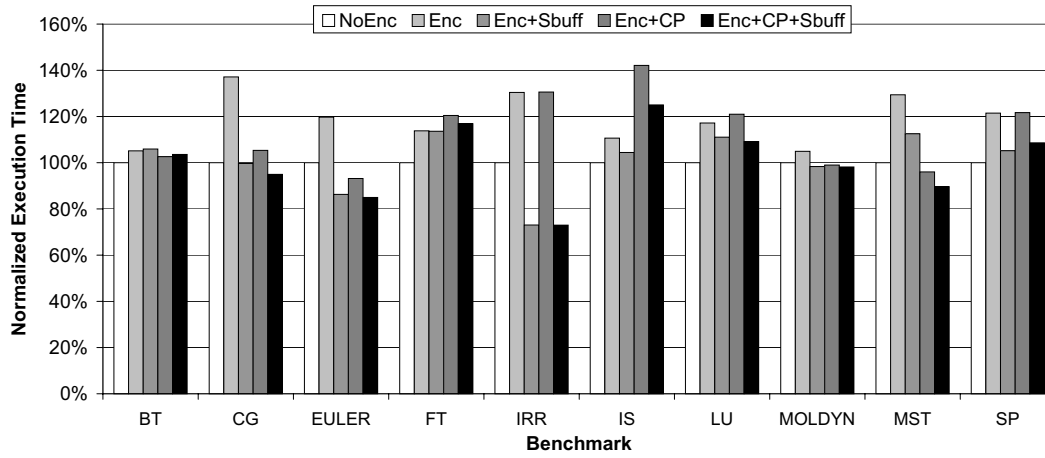


Figure 4: Non-SPEC app. performance with different predecryption schemes.

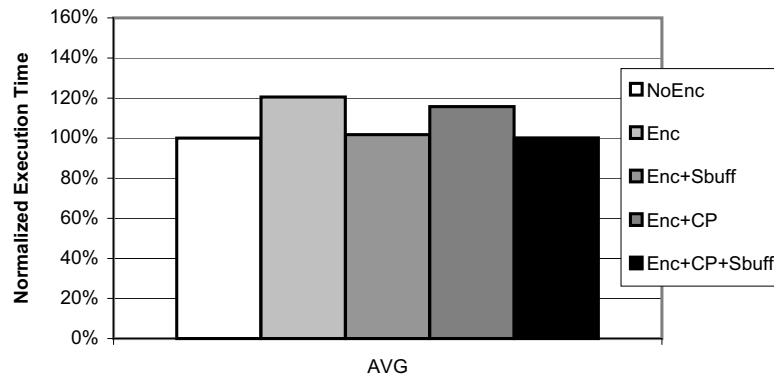


Figure 5: Average performance of all apps. with different predecryption schemes.

ryption. If a prefetcher is only able to hide memory latency alone, the relative execution time in $Enc+CP+Sbuff$ would be higher than that in $NoEnc+CP+Sbuff$. If a predecryption is equally good at hiding decryption latency as it is at hiding memory latency, the relative execution time in $Enc+CP+Sbuff$ would be similar to that in $NoEnc+CP+Sbuff$. Finally, if the prefetcher is better at hiding decryption latency than it is at hiding memory latency alone, the relative execution time $Enc+CP+Sbuff$ would be lower than that in $NoEnc+CP+Sbuff$.

Our results in Figures 6-8 indicate that in most applications the prefetcher is at least as good at hiding decryption latency as it is at hiding memory latency, and in some applications it is noticeably better. This indicates that the prefetcher is indeed hiding some of the decryption latency in addition to the memory latency. In fact, the fraction of decryption latency that the prefetcher hides is often larger than the fraction of the memory latency it hides. Overall, without encryption, a prefetcher reduces execution time by 12% on average, while the reduction is 16% in a system with memory encryption. This indicates that predecryption is a promising scheme for alleviating the performance impact of memory encryption.

However, one disappointment in our current setup is the lack of significant benefit from adding a correlation predictor to the stream buffers mechanism. To investigate this, we evaluate the performance of predictors in different configurations, and show the results in Figures 9, 10, and 11.

Each bar in these figure has three segments. The bottom segment shows the number of L2 cache misses that are correctly predecrypted with the given predecryption setup. The

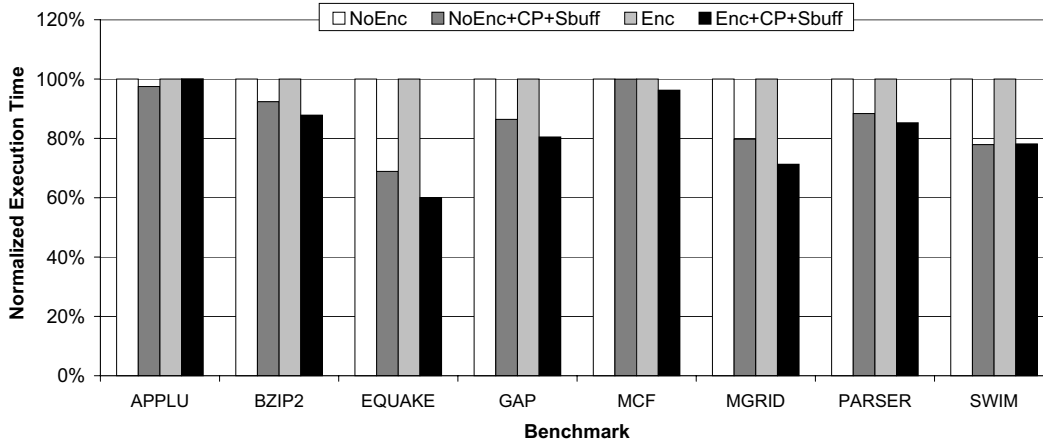


Figure 6: Predecryption benefit over standard prefetching for SPEC2K apps.

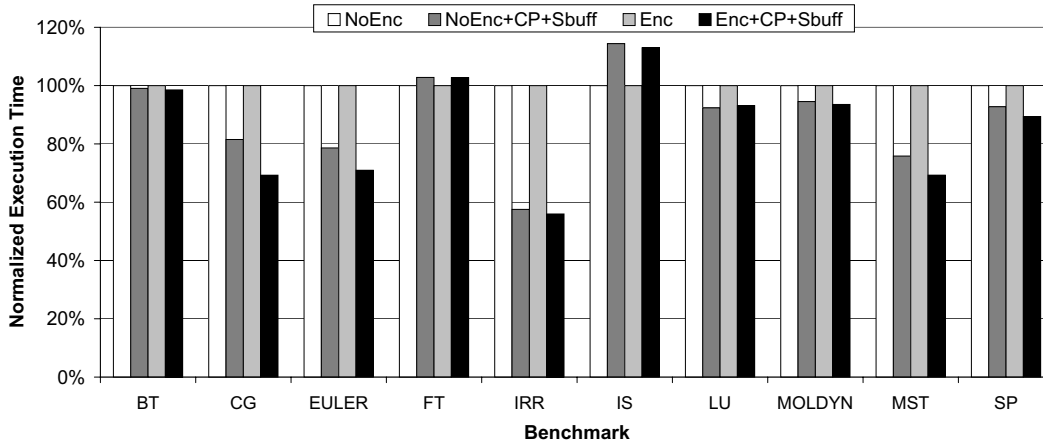


Figure 7: Predecryption benefit over standard prefetching for non-SPEC2K apps.

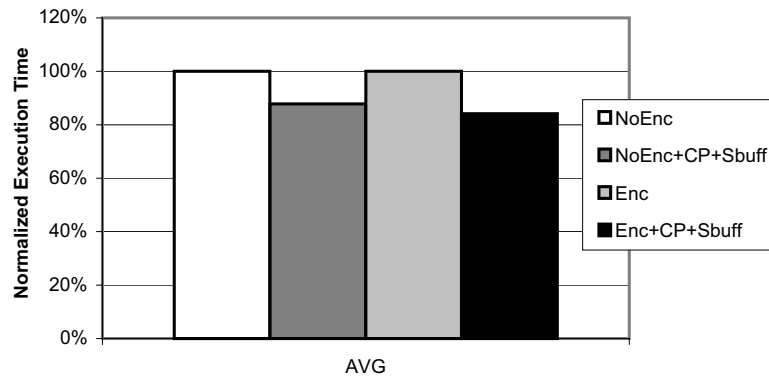


Figure 8: Average predecryption benefit over standard prefetching for all apps.

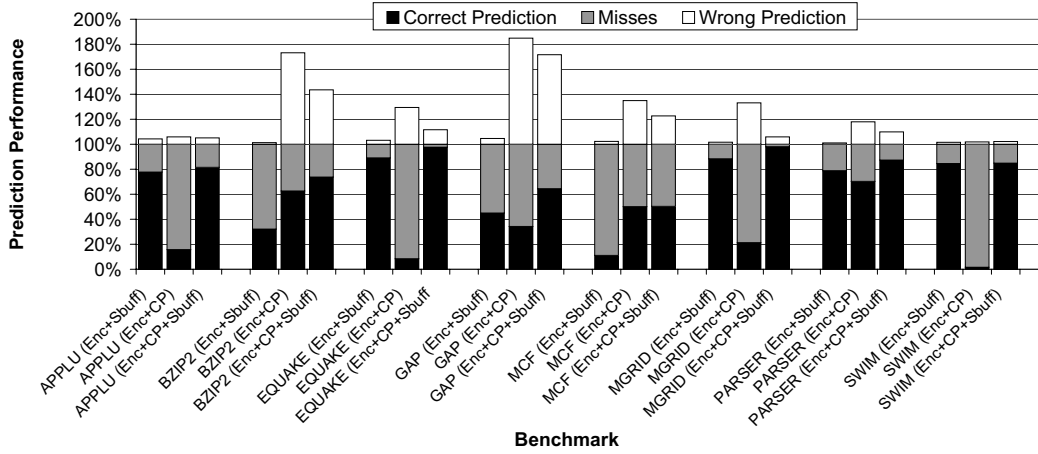


Figure 9: Prediction performance of SPEC2K apps. for different predecryption schemes.

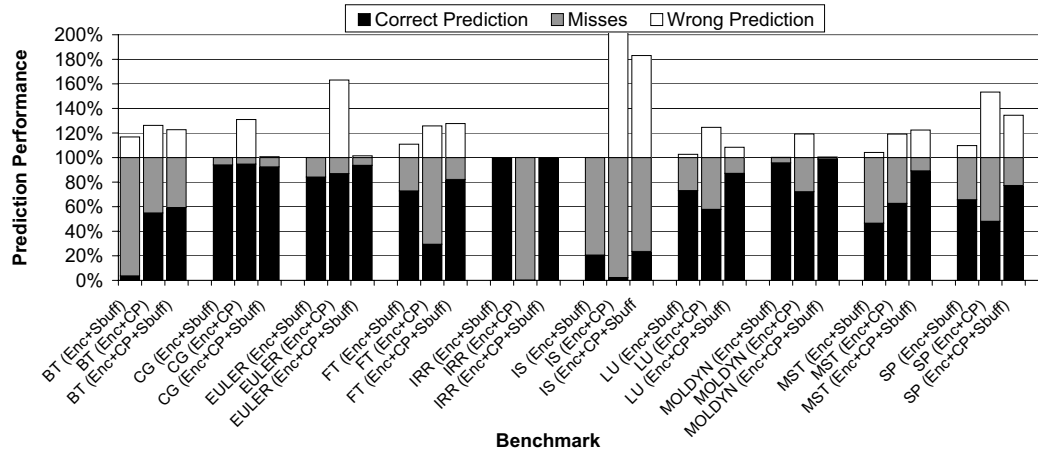


Figure 10: Prediction performance of non-SPEC2K apps. for different predecryption schemes.

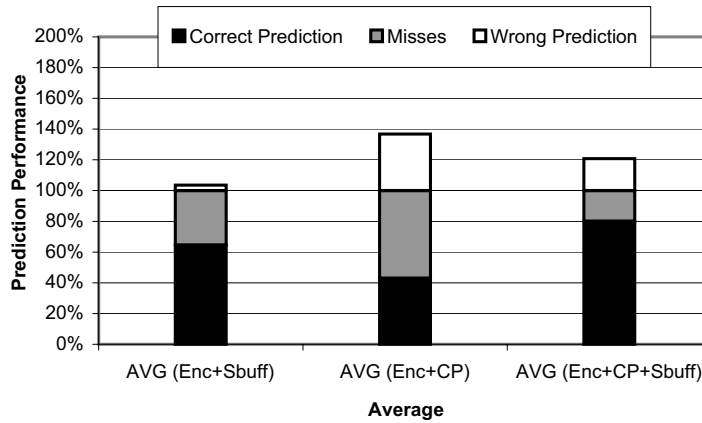


Figure 11: Average prediction performance of all apps. for different predecryption schemes.

full or partial miss latency (which includes memory latency and decryption latency) of these accesses is hidden and the data is read from the predecryption buffer. The middle segment shows the percentage of L2 cache misses that go to the main memory and also suffer the full decryption latency. In other words, this is the percentage of L2 cache misses that do not benefit at all from predecryption. The top segment shows the number of blocks that are predecrypted needlessly, meaning that the blocks are either never needed, or they are replaced from the predecryption buffer before they are actually requested by the processor. All bars are shown relative to the number of L2 misses in the baseline configuration. On average, the correlation predictor alone successfully predecrypts about 43% of the original L2 cache misses. However, this scheme also causes an extra 37% more predecryptions of data that is not useful to the application. These extra predecryptions could be the reason for the lack of performance benefit when the correlation predecryptor is added to the stream buffers. These extra memory accesses cause contention delays and interfere with memory requests for valid data. This is clearly evident in the *is* benchmark, where the addition of correlation prefetching slows down the execution of the benchmark, due to a high number of wrong predictions and low number of correct ones.

Future work will involve implementing a confidence scheme to only send predecryption requests for which there is some confidence that the data will be used. Another reason for low benefit of correlation prefetching is that the table may be too small for the data set size in some applications. We use a tagged correlation table and the lack of attempted predecryptions in some applications (e.g. *irr*) indicates that many misses fail to

match an entry in the correlation table. However, a larger table may not be feasible to put on-chip, and our future work will focus on improving the table’s hit rate while maintaining or further reducing its size. Our results show that stream buffers are very accurate in terms of predecrypting only useful data, and they also act as a filter that helps utilize the correlation table better. The combination of the two prediction mechanisms can predecrypt a very high percentage of L2 cache misses (80%), while only generating 20% additional unnecessary predecryptions, as shown in Figure 11.

Another interesting observation that can be made from Figures 3 and 4, and Figures 9 and 10 is that in four benchmarks (applu, bt, ft, and mcf), we correctly predecrypt a large portion of L2 cache misses, but the execution compared with *Enc* is still similar. This is because many of the correct predecryptions in these applications only hide partial miss latency, and a significant fraction of the latency remains. This, combined with the increase in bus traffic due to mispredictions (top segment of each bar in Figures 9-11), results in a low overall benefit from predecryption. This confirms that reducing these extra predecryptions should make our scheme even more attractive.

Finally, since our predecryption approach utilizes prefetchers, we compare the overhead of predecryption to a system that has comparable prefetching in Figure 12. The bars are normalized to the *NoEnc+CP+Sbuff* case. These results show that in most cases, our predecryption approach has a reasonable overhead even when compared to a system that already has prefetching. On average this overhead is only around 15%. This result relates back to Figures 6- 8 where we showed that predecryption not only hides memory latency,

but by prefetching far ahead we are able to hide decryption latency as well. We find this result promising because with many processors today already having prefetching hardware, it may be possible to tune them to prefetch further ahead to support direct memory encryption with a reasonable overhead.

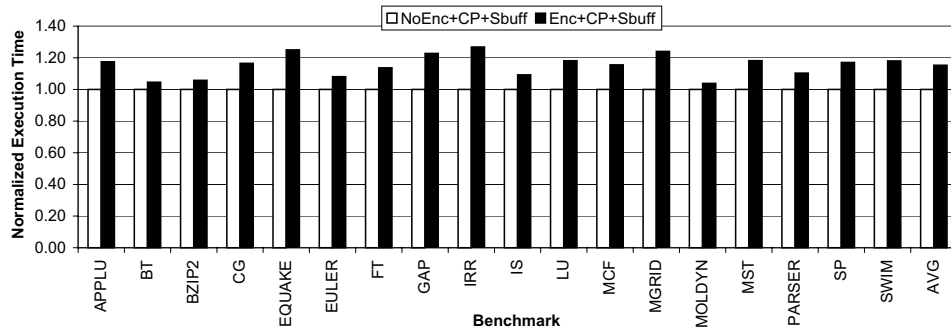


Figure 12: Predecryption overhead for all benchmarks compared to a system with prefetching.

6. Conclusions

Our results show that predecryption compares favorably to schemes such as one-time-pad because the execution time of some benchmarks can actually be reduced even with the extra latency of memory encryption. This is due to the fact that in the worst case scenarios of predecryption and one-time-pad, the latency overhead is the same, but the best case latency of predecryption can be much lower than the best case latency of one-time-pad for an L2 cache miss. Additionally, in schemes such as one-time-pad there will always be some slowdown in execution time because the decryption delay of a memory request can never be fully hidden as it can in our scheme. Since the data shows that the correlation predecryptor does not provide much performance improvement over simply using stream buffers alone, we feel that this scheme will appear even more attractive once the correlation predecryptor implementation is optimized.

Our predecryption scheme also benefits from the fact that known encryption algorithms can be used directly on the data. This could be desirable from a security point of view because well-studied algorithms with known security strengths can be used, allowing more control over the strength of the memory encryption scheme. Lastly, the small amount of communication that would be needed (a one-time passing of a key) for a multiprocessor system makes this predecryption scheme more easily extendable to multiprocessor environments.

References

- [1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *the Second International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.
- [2] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. *Technical Report EE-CEG-95-1, Cornell University*, February 1995.
- [3] T. F. Chen and J. L. Baer. Reducing Memory Latency via Non-Blocking and Prefetching Cache. In *the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.
- [4] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [5] T. Gilmont, J-D. Legat, and J-J. Quisquater. Enhancing the Security in the Memory Management Unit. In *Proc. of the 25th EuroMicro Conference*, 1999.
- [6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (First Quarter), 2001.
- [7] A.B. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, San Francisco, CA, 2003.

- [8] IBM. *IBM Power4 System Architecture White Paper*, 2002. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>.
- [9] International Planning and Research Corporation. *Sixth Annual BSA Global Software Piracy Study*, 2001. <http://www.bsa.org/resources/2001-05-21.55.pdf>.
- [10] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *the 24th International Symposium on Computer Architecture*, pages 252–263, June 1997.
- [11] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.
- [12] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *the 28th International Symposium on Computer Architecture*, pages 144–154, June 2001.
- [13] D. Lie, J. Mitchell, C.A. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *IEEE Symposium on Security and Privacy*, 2003.
- [14] D. Lie, C.A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

- [15] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.
- [16] P.R. Schaumont, H.Kuo, and I.M. Verbauwhede. Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor. In *Design Automation Conference*, 2002.
- [17] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *the 33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.
- [18] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *29th International Symposium on Computer Architecture (ISCA)*, May 2002.
- [19] Standard Performance Evaluation Corporation. Spec benchmarks. <http://www.spec.org>, 2000.
- [20] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.
- [21] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.