General Adaptive Replacement Policies

Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
yannis@cc.gatech.edu

ABSTRACT

We propose a general scheme for creating adaptive replacement policies with good performance and strong theoretical guarantees. Specifically, we show how to combine any two existing replacement policies so that the resulting policy provably can never perform worse than either of the original policies by more than a small factor. To show that our scheme performs very well with real application data, we derive a virtual memory replacement policy that adapts between LRU, loop detection, LFU, and MRU-like replacement. The resulting policy often performs better than all of the policies it adapts over, as well as two other hand-tuned adaptive policies from the recent literature.

1. INTRODUCTION

Replacement policies play an important role in memory management. Caching mechanisms at every level of a storage hierarchy (processor cache, VM and file system cache, web cache, etc.) are of primary importance for high performance, and the choice of replacement policy can make a significant difference in the cache efficiency. Unfortunately, most replacement policies in use have well-known failure scenarios. For instance, a recency-based policy, like LRU, fails badly for loops slightly larger than memory. A frequency-based policy, like LFU, does badly when different parts of memory have different and time-variant usage patterns.

In this paper we formulate and evaluate the idea of *general adaptive replacement policies*. We propose that it is easy to fix any particular replacement policy, by combining it with a different policy and adaptively switching between the two. Although many past replacement policies try to adapt to current behavior, in this paper we offer a general adaptivity scheme (one may prefer to call it *meta-adaptivity*) with both good theoretical properties and good performance with real program data.

Specifically, given two replacement policies A and B, we can derive an adaptive policy AB that will never incur more than a small multiplicative constant (e.g., 2 times) as many faults as either A or B. (The scheme can be straightforwardly generalized to more than 2 policies by adapting between AB and a third policy C, etc.) Although a 2x bound may seem unimpressive at first, it is quite low as far as worst-case guarantees are concerned, and for real program data the behavior is significantly better. The worst-case bound holds even though A and B can differ drastically. That is, there can be inputs for which policy A will incur many times (up to the memory size M, which can be from many tens of thousands to a million for real systems) as many faults as B and vice versa. Nonetheless, the adaptive policy AB will never be much worse than either A or B, essentially adopting the good characteristics of both.

The practical benefit of this result is as a powerful weapon in the replacement policy designer's arsenal. It is very hard to design good replacement policies, even when the workload locality characteristics are well-known. For example, it is hard to design a good replacement policy that a) behaves comparably to LRU in replacing pages that were not recently accessed; b) eagerly replaces pages that are accessed only once, similarly to LFU; and c) behaves optimally for large linear loops. In contrast, it is easy to find simple policies (LRU, LFU, loop detection) that capture a single locality characteristic each (recency, frequency, or optimal behavior for linear loops). Our approach gives an effective and general way to combine such simple, special-purpose policies to create a good general-purpose policy. Furthermore, our result can be viewed as a generalization of existing adaptive approaches: policies such as EELRU [17] can be seen as specific instantiations of our general adaptivity scheme.

The idea (and theory) of general adaptive replacement policies makes no restrictive assumptions about the policies A and B or the domain. Therefore, our adaptivity scheme can be used in any cache management domain, subject to technology constraints. In this paper, we specifically evaluate adaptive policies in the context of virtual memory page replacement by performing simulations with a large number of traces from previous studies. We show that our adaptive policy performs very well with real program data, often outperforming LRU (as well as all the other policies it adapts over) by more than 40%. Compared to recently proposed, hand-tuned adaptive algorithms, such as SEQ [6] and EELRU [17], our general adaptive approach commonly performs better while being much simpler, both conceptually and in terms of the tuning effort required.

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

2. PRINCIPLES OF ADAPTIVE POLICIES

2.1 Background, Motivation and Impact

The main limit result in the area of replacement policies was shown by Sleator and Tarjan [16]. They proved that any deterministic on-line replacement policy performs arbitrarily worse in the worst case than the optimal off-line replacement policy (OPT). More specifically, for every on-line replacement policy A, there are reference sequences such that A incurs at least M (the memory size in blocks) times as many faults as OPT does for the same amount of memory. Such results created the area of *competitive analysis* of algorithms.

The LRU (Least Recently Used) replacement policy is considered the standard benchmark for replacement because of its good performance in practice. In terms of competitive analysis, LRU is optimal: it incurs *at most* M times as many faults as OPT for any sequence, matching the above lower bound. That is, no policy can perform better than LRU relative to OPT (although many can match it, in terms of competitive analysis). Additionally, a second result establishes that LRU with twice as much memory available will perform at most twice as badly as OPT.

We next use competitive analysis techniques to derive a general adaptivity scheme: given two replacement policies A and B, we derive a policy AB that never incurs more than two (or a small constant, in the general case) times as many faults as either A or B. This result can be viewed as a way to combine two policies and get the best elements of both. An alternative way to see our scheme, however, is as a general way to improve a given policy A for arbitrarily many cases. That is, for a given policy A, we can specify an ad hoc policy B that recognizes some of the bad cases for A and performs well in these cases (and arbitrarily badly in all other cases). Then, by creating an adaptive policy combining A and B, we get a policy that, from the perspective of competitive analysis, is at least as good as A in all cases, and much better in the cases covered by B. Thus, although we can never approach the performance of OPT for all inputs, we can get close for arbitrarily many inputs. This is an important complement to the competitive analysis results discussed above.

2.2 Adaptive Replacement Scheme

In this section we will use standard replacement policies terminology. The replacement algorithm is managing a *buffer* (a.k.a. a *memory*) that holds a finite number of equal-sized *blocks* (a.k.a. *pages*). A reference to a block not in the buffer is a *fault* (a.k.a. a *miss*) and causes the referenced element to be added to the buffer. If the buffer is full, an existing block must be *evicted/replaced* to make room for the new one. A reference to an element in the buffer is a *hit*. We call the input to a replacement policy a *reference sequence*. Let M be the size in blocks of the memory being managed. When we compare algorithms, we implicitly assume they are all acting on memories of the same size.

The goal of an adaptive policy is to adaptively (on-line) switch between policies A and B so that it uses B whenever recent behavior indicates that B would outperform A. We will prove that an adaptive replacement policy AB will never perform more than a constant multiplicative factor worse than either A or B. We call this property *robustness*.

DEFINITION 1. *Robustness: A replacement policy $R1$ is c-robust with respect to replacement policy $R2$ if $R1$ can never incur more than c times as many faults as $R2$ for any input and memory size.*

There are multiple ways to define an adaptive policy—we will next present a very simple definition that results in a straightforward proof, but is not ideal for actual implementations. Later, we will add more logic to the definition, without sacrificing our proof guarantees.

For any two replacement policies A and B, we define an adaptive replacement policy AB. AB simulates both A and B in memory. AB can tell at any point in time what the state and behavior of A and B would be—i.e., what blocks policies A and B would hold in memory, whether a reference would be a hit or a miss, and what block they would replace. (In the following, we use the present tense for the actions that A/B would perform at the same point in the input sequence, e.g., we write "a reference is a fault for A", instead of "a reference would be a fault for A".) AB then picks to imitate the behavior of either A or B:

DEFINITION 2. *Adaptive Replacement Policy AB: At every fault, replace blocks as follows.*

- *if the reference is a fault for A but not for B, evict one of the memory blocks that are not in B's memory. (There have to be such blocks, or the reference would have been a fault for B since the memories of AB and B have the same size.)*

  (We call this case "AB is in 'B' mode".)
- *otherwise*
  - *if the memory contains blocks that are not in A's memory, then evict one of those blocks.*
  - *otherwise evict whichever block A evicts.*

  (We call this case "AB is in 'A' mode".)

To prove the robustness of the adaptive policy AB with respect to A and B, we first need a simple lemma:

LEMMA 1. *Consider the same reference sequence processed by replacement policies A, B and AB. If at a certain point a block is both in the buffer managed by A and in the buffer managed by B then it is also in the buffer managed by AB.*

Proof: The property holds initially and if it holds up to a point in the reference sequence, then consider the next fault for either policy A or B. If it is not a fault for AB, then the property will hold after the replacement (because the new block is already in AB's memory). If it is also a fault for policy AB, then there are 3 cases in the AB eviction logic:

- AB evicts a block not in B's memory.
- AB evicts a block not in A's memory.
- AB evicts the same block as A

In each of the above cases, the block evicted by AB could not have belonged in both A's and B's buffer before the eviction or will be evicted from one of them at the same time. Thus, if the property held before the current fault, it will hold after the eviction. □

The lemma effectively says that the set of blocks held in AB's buffer is a superset of the intersection of the sets of blocks in A's and B's buffers at all points in time.

We can now prove the main theorem for adaptive replacement.

THEOREM 1. *The adaptive replacement policy AB is 2-robust with respect to both A and B. That is, AB, as defined above, never incurs more than twice as many faults as either A or B.*

Proof: We define two "potential" quantities and examine how their values change on every fault for either policy A or B.

Let d_A be the number of blocks currently in AB's buffer that are not in A's buffer at the same point in the execution.

Let d_B be the number of blocks currently in AB's buffer that are not in B's buffer at the same point in the execution. The above values of d_A, and d_B change as follows on every fault (we denote the new values d'_A, and d'_B):

(Note that according to Lemma 1 a hit for both A and B implies a hit for AB and none of the potentials changes.)

1. fault for B, hit for A, hit for AB: $d'_A = d_A$, $d'_B \leq d_B$
2. fault for B, hit for A, fault for AB: $d'_A < d_A$, $d'_B \leq d_B + 1$
3. fault for B, fault for A, fault for AB: $d'_A \leq d_A$, $d'_B \leq d_B + 1$
4. fault for B, fault for A, hit for AB: $d'_A \leq d_A$, $d'_B \leq d_B$
5. hit for B, fault for A, hit for AB: $d'_A \leq d_A$, $d'_B = d_B$
6. hit for B, fault for A, fault for AB: $d'_A \leq d_A + 1$, $d'_B < d_B$

We do not show all the case-by-case reasoning needed to derive the above values since the derivation is tedious but straightforward. As a single example, consider case 2. In this case, there is a fault for B and AB but a hit for A. Thus AB is in "A" mode and furthermore AB's buffer contains blocks not in A's buffer (otherwise A would have suffered a fault as well). Then, the newly referenced block is already in A's buffer and AB will replace a block *not* in A's buffer. Thus, the difference of the two buffers will strictly decrease: $d'_A < d_A$. At the same time, AB and B will fault-in the same block, but they may pick two different blocks to evict and these blocks could have been common to both their buffers before. Thus d_B (the number of blocks in AB's buffer that are not in B's buffer) may increase but at most by one (hence the $d'_B \leq d_B + 1$).

Now we are ready to show the first part of our result. AB is 2-robust relative to A. To see this, consider that for each fault of AB that is not a fault of A (case 2) d_A is strictly decreasing (by 1). But d_A is originally 0, has to stay non-negative, and increases only in the case where both A and AB incur a fault (case 6). Thus, for every fault of AB that is not a fault of A, A and AB must have suffered a common fault earlier. That is, AB can only incur up to twice as many faults as A.

Similarly, we can show that AB is 2-robust with respect to B. If AB suffers a fault that is not a fault of B (case 6) then d_B is strictly decreasing. But d_B is originally 0, has to stay non-negative, and increases only in cases 2 and 3, where both B and AB incur faults. Thus, for every fault of AB that is not a fault of B, B and AB must have suffered a common fault earlier. That is, AB can only incur up to twice as many faults as B. □

The interesting aspect of the above theorem is not its sophistication but its simplicity. As our adaptive scheme shows, it is really easy to produce an adaptive policy that satisfies the desired theoretical guarantees. (Nevertheless, we queried multiple experts and none were aware of the above result.)

An adaptive scheme can be used recursively. We can, for instance, adapt between a regular replacement policy, such as LRU, and another adaptive replacement policy, such as an MRU-LFU combination. In this way, we can implement adaptivity among an arbitrary number of existing policies. Nevertheless, the order and structure of the composition matter, both in terms of practical performance and in terms of theoretical guarantees. For instance, if we want to adapt among three policies A, B and C, we get different properties by composing them as $(AB)C$ (i.e., adapting between A and B and then adapting between the resulting policy and C) than by composing them as $A(BC)$. Based on Theorem 1, the latter algorithm is 2-robust relative to A, while the former is only guaranteed to be 4-robust. (We believe that it is possible to derive tighter bounds for adapting among more than two policies, however.)

2.3 More Sophisticated Adaptation

We can add several elements to the above basic adaptivity scheme without affecting the robustness result. Notably, AB can remember the last k faults of either A or B and imitate the policy with the fewest recent faults. Also, the adaptive policy can remember either all recent faults or just faults for either A or B but not for both. Specifically, an interesting general adaptation scheme is the following:

DEFINITION 3. *Adaptive Replacement Policy $AB(k)$:*

Let k be a constant parameter to the policy. Let s be the number of faults for policy A among the k latest faults for either A or B but not both.

On every fault, $AB(k)$ replaces blocks as follows:

- *if $s > \frac{k}{2}$ ("$AB(k)$ is in 'B' mode")*
  - *if B also suffers a fault on this reference and it evicts a block that is currently in AB's memory, then evict the block that B evicts*
  - *otherwise, evict one of the memory blocks that are not in B's memory.*
- *if $s \leq \frac{k}{2}$ ("$AB(k)$ is in 'A' mode")*
  - *if A also suffers a fault and it evicts a block that is currently in AB's memory, then evict the block that A evicts.*
  - *otherwise, evict one of the memory blocks that are not in A's memory.*

That is, just as before, the adaptive policy simulates both A and B in memory and picks to imitate the behavior of one of them—in this case the one with the fewest faults among the k most recent ones. We can prove an adapted version of Theorem 1 for the more complex adaptive policy $AB(k)$. Specifically, we can show that $AB(k)$ is 3-robust relative to either A or B. (The 3 bound is probably not tight, and for many specific instances of A and B it can be shown that $AB(k)$ is 2-robust relative to both.) We do not show the proof of this theorem, as it is significantly more convoluted than the proof of Theorem 1 and the insights it offers are low-level (i.e., only useful to readers who attempt to prove similar theorems). The complication is in the potential functions, which now also need to model the "memory" of past events.

This sophisticated version of adaptivity is the one we use later in our experiments. Thus, when we talk about adaptive policy AB in the rest of this paper, we really refer to the general form $AB(k)$.

2.4 From Policy to Algorithm to Mechanism

One of the prime contributions of Computer Systems research has been the distinction of the concepts of *policy*, *algorithm*, and *mechanism*. So far in this paper, we have only talked about replacement policies, but we have not discussed how these can be reified in algorithms and implemented (exactly or approximately) in systems mechanisms. We address this topic next.

There are three elements of the adaptive algorithm that add space overhead: first, algorithm AB needs to maintain data structures that represent the contents of the memories of both A and B at the same point in the execution. (Of course, these are not the full contents of the memories of A and B but just data structures containing the identifiers of blocks in memory.) Second, algorithm AB needs to keep whatever information A or B would keep in order to simulate these algorithms. Third, AB needs to have a memory of the last k faults for either A or B but not both. The third overhead is minimal. For $k = M$, for instance, the algorithm only needs to maintain a bit vector of size M bits, two counter variables (to keep the current sums of zero and one bits) plus a pointer (to show the vector's current end). The second overhead (maintaining the data kept by either A or B) can often be eliminated if A and B keep the same information (e.g., recency or frequency) for their replacement decisions and just use this information differently. Similarly, the first overhead can often be minimized in practice. In the worst case, it may seem that AB needs to maintain three times as much information: the memory contents of A, those of B and those of AB. Nevertheless, since the memories of A, B, and AB typically have a high content overlap, we get space savings by only keeping a single data structure of all blocks that are in one of the three memories.

Of course, an adaptive replacement policy AB has some time overhead, as it needs to simulate both policies A and B. Nevertheless, in practice, the time cost of executing a replacement algorithm is either small or unbearable! The distinction is linked to when the algorithm needs to perform work. The standard rule of thumb is that an algorithm is *realizable* if it only performs work on very infrequent events (like memory misses) instead of on every reference. Thus, for instance, policies such as LRU and LFU are not realizable for processor caches or virtual memory systems because they require information to be updated on every reference (and not just on every fault). Instead, LRU and LFU are typically approximated by realizable algorithms that ignore some high frequency information—e.g., as in the case of a CLOCK algorithm. An excellent property of our adaptivity scheme is that if replacement policies A and B are realizable, replacement policy AB is also realizable. Specifically, the adaptivity mechanism only needs to perform actions when a fault is suffered either by the adaptive algorithm or by A or by B. These events are very rare.

In terms of specific applications, adaptive replacement algorithms look particularly promising at the level of virtual memory page replacement and file system caching. One of the reasons is that current virtual memory mechanisms place a high premium on "thrashing avoidance". That is, we want a good replacement algorithm to avoid its worst-case behavior and not cause system thrashing for common memory access patterns. This is hard to do when a single policy is used for all applications and memory sizes. Nevertheless, adaptivity can help the replacement algorithms behave gracefully in a much larger set of circumstances.

3. RELATED WORK AND DISCUSSION OF THE AREA

Having finished the description of our adaptivity scheme, we now discuss its relation to previous work in the literature. There is a huge volume of work on replacement algorithms, so our presentation needs to be selective. We concentrate on adaptive replacement algorithms for RAM (file system or virtual memory caching) and on recent work with good lists of further references.

Adaptive replacement is certainly not a new idea—even some of the oldest results in replacement (e.g., the Atlas

loop detector [2]) can be thought of as policies that adapt to the current behavior of the system. Several policies have been explicitly called "adaptive". These include SEQ [6], EELRU [17, 18], DEAR [3], LRFU [11, 10], FBR [15], LIRS [7], ARC [13], and more. Our work is different from all of the above in that it tries to offer very general adaptation techniques with both good theoretical properties and good practical behavior. As a general rule, past policies offered no guarantees of worst-case behavior relative to the policies they adapted over. (An exception is EELRU, for which it is shown [18] that it is 3-robust relative to LRU—i.e., it will never incur more than 3 times the faults of LRU.) Furthermore, the adaptivity of past schemes was limited to a very specific policy, as opposed to being applicable to any replacement policy.

There are two main patterns emerging from the long list of adaptive replacement algorithms in the recent literature. The first is *loop detection* and the second is *mixing frequency and recency*. Many algorithms combine both patterns to some extent. Loop detection for replacement purposes has been proposed several times in order to address the shortcomings of LRU for large scale looping patterns. The well-known Atlas loop detector [2] is commonly cited as the predecessor of more recent loop detection work (e.g., [14, 6]). Plain loop detection is not usually done in any modern system and is generally considered an artifact of the past, when large, linear array manipulations dominated the space of expensive computations. A different "fashion" has emerged in the last decade: often motivated by large multimedia processing tasks that only touch the blocks of a stream once, many replacement algorithms try to differentiate between regularly accessed blocks and single-use blocks that should be evicted very soon. Much of the replacement work in the Linux kernel has originated from such concerns. The easiest way to get this differentiation of pages is by combining recency and frequency information for blocks. Although the idea of combining recency and frequency is much older (e.g., FBR [15]) it is a common thread in algorithms like DEAR [3], LRFU [11, 10], LIRS [7], and ARC [13].

It is interesting to classify replacement algorithms depending on whether they are applied to the file system cache (i.e., the portion of RAM that stores pages from files other than the swap file), to the disk controller cache, or to virtual memory page caching. Algorithms like SEQ [6] and EELRU [17] primarily target virtual memory management, while algorithms such as DEAR [3], LRFU [11, 10], FBR [15], LIRS [7], and ARC [13] are more targeted towards file system data caching, either at the OS or the disk controller level. The distinction is very important for two reasons. First, the patterns of access for the two kinds of blocks are different. For instance, explicit file system traffic often exhibits more regularity (spatial locality) but lower temporal locality than virtual memory accesses. This means that LRU is harder to "beat" in virtual memory applications (for representative workloads) than for file system caching. Even more importantly, however, file system caching studies often do experimental evaluation at the level of the traffic that the disk controller sees, and not at the level of all accesses to blocks. This means that a lot of the information is discarded and what is being evaluated is not the policy at hand but the combination (in a filtered or *segmented queue* [20] way) of whatever replacement algorithm already exists in lower-level caches together with the policy at hand. For example, the virtual memory traces used in the SEQ [6] and EELRU studies [18] are 30-1000 times the size of the file access traces used in the LIRS [7] and LRFU [11] studies! (This is true even though the former traces are already reduced to filter out high-frequency references and the two kinds of traces have similar footprints—i.e., the number of distinct pages touched is usually the same or lower.) We believe that virtual memory traces are much better suited to evaluating replacement policies than file system traces. Once the right policy is found, then the reification of this policy for a given hardware configuration and OS limitations should take place. File system traces are acceptable for evaluating policies at the "external" level, if most of the OS memory management mechanisms are considered a given.

Much work on replacement algorithms has concentrated on providing replacement guarantees under some mathematical modeling assumptions regarding block references. For instance, Markov modeling has often been used as the basis for replacement work [4, 5] and optimal online replacement algorithms under Markovian assumptions can be derived [9]. A 0-th order Markov model in the address domain corresponds to the well-known *independent reference model* [1], while in the recency domain it corresponds to the *LRU stack model* [19]. In practice, mathematical modeling has not proven to be very promising for producing good online replacement algorithms. Our adaptivity scheme offers worst-case guarantees, while making no assumption about either the memory reference behavior or the replacement algorithms that it adapts over.

4. EXPERIMENTAL MEASUREMENTS

4.1 Simulated Workloads and Settings

To evaluate our adaptivity scheme experimentally, we used the extensive set of traces previously used in the EELRU study [18] and simulated them for the same memory ranges as that study. Eight of these traces were also used in the study of the SEQ algorithm [6]. Five of the rest are traces of SPEC95 benchmark applications from the Etch traces collection [12]. The final six traces are small-scale programs that mostly serve as sanity-checkers since they were not intended to exercise VM functionality. The EELRU study discusses the locality characteristics of all traces and argues that they are a good set for virtual memory experiments. We will not describe here the traces in detail, but briefly:

- the eight traces from the SEQ study are for "applu" (a PDE solver), "gnuplot" (a postscript graph generator), "ijpeg" (an image conversion utility), "m88ksim" (a microprocessor simulator), "murphi" (a protocol verifier), "perl" (a scripting language), "trygtsl" (a matrix calculation program), and "wave5" (a plasma simulation)

- the five Etch traces are for "go" (an AI program for the game of Go), "CC1" (the gcc compiler core), "compress" (a compression utility), "perl" (same application as before but with a different input), and "vortex" (a database program)

- the six small-scale traces are of "espresso" (a circuit simulator), "gcc" (the C compiler), "ghostscript" (a PostScript engine), "grobner" (a formula rewriter), "lindsay" (a communications simulator for a hypercube) and "p2c" (a Pascal to C compiler)

- the Etch trace collection also contains more traces for Windows applications (Word, Powerpoint, Netscape, Photoshop, Acrobat Reader). These are older versions of the applications and the inputs are not such as to exercise large-scale memory consumption (the Etch traces are mostly intended for processor cache studies). Nevertheless, we simulated these traces, as well, to ensure that our adaptive scheme does not underperform LRU in any case.

In our experiments, we selected several policies that capture distinct locality characteristics and simulated an adaptive algorithm over all of them. Adapting over many policies is a good experiment because it tests whether the resulting algorithm adapts quickly enough to capture the benefit of all individual policies, yet is stable enough to not be fooled by their pathological cases. Our algorithm adapts among 8 different policies: 5 different MRU-LRU combinations (i.e., policies that evict the c-th MRU page, for 5 different values of c, spaced equally from the minimum size permissible by the traces to the memory size), LRU, a loop detector (LD), and LFU. The 5 different MRU-LRU combinations are intended to emulate the "early eviction" capability of a policy like EELRU. The rest of the policies (LFU, LRU, LD) capture distinct locality characteristics. (Our loop detector is a simple heuristic that observes a constant number (10 in our case) of references to consecutive blocks and does MRU replacement on the least recently accessed sequence.) As mentioned previously, the exact way of composition (e.g., order) affects the behavior of the adaptive algorithm, since our general scheme adapts over two policies, and then we adapt over the resulting policy and a third one, etc. The composition we tried was linear in the above order (MRU[0..4], LRU, loop detector, LFU). Nevertheless, we also tried two other permutations that yielded almost identical results.

We should note that our LFU and loop detector are not fully faithful to what their descriptions suggest. The traces used in our simulations were already reduced with the OLR algorithm [8], to make their size manageable. This means that the traces provide for fully accurate LRU simulations (for the memory sizes shown), but in order to simulate any other policy accurately, the policy has to keep the m most recently accessed pages in memory (m is a constant fixed for each trace) and ignore accesses to them. This means, for instance, that our LFU simulation is really a simulation of a policy just like LFU, but not for the few most recently accessed pages. In essence, our "LFU" is handicapped (since it receives less information than what true LFU would) but closer to what would be implemented in practice (since realizable replacement algorithms ignore accesses to recently used pages anyway).

The simulation framework we used follows established experimental practices in the virtual memory management literature. Specifically:

- The simulation is for a single program trace at a time. This allows us to see whether the replacement policy captures the locality features of the program. Although real systems execute multiple programs together, there are standard arguments why the effects of process scheduling should not be integrated in a replacement policy simulation. First, the results are very specific to the process scheduling being simulated and almost never reflective of reality: page replacement is not independent of process scheduling—instead it often completely dictates it (i.e., a new process starts executing only when a fault is incurred). Second, the replacement problem for multi-process workloads can be decomposed into a problem of assigning (and dynamically adjusting) space to each process and then performing replacement within the process space. Third, in many of the interesting practical scenarios where paging is an issue, a single process consumes the vast majority of virtual memory.

- The simulated memories range from very small to very large sizes. At the low end, the memory size is such that the program would spend most of its time paging. At the high end, the program does not page at all. This choice often seems strange to readers unfamiliar with simulation experiments in virtual memory management: after all, most programs would not even be started if they were to spend most of their time paging. Nevertheless, it is commonly accepted that a good replacement algorithm is one that behaves fairly well for any amount of available memory. In practice, the available memory ranges dramatically due to hardware configuration (the same OS can be used on machines that have from a few MB to many GB of RAM) and other programs running.

- The simulated policies are the idealized policies (e.g., LRU) and not their realizable versions (e.g., CLOCK, or a FIFO-LRU segmented queue). In this way, it is the policy itself that is being evaluated relative to its ability to capture the locality characteristics of the workload. Subsequently, appropriate approximations of the policy can be derived. Typically these approximations are fairly accurate and efficient, although their derivation may require significant engineering effort.

Finally, it is worth pointing out that our simulations were performed with *extremely low* amounts of tuning. The results we show are practically the very first we got. The only parameter of our adaptivity is the number k of recent faults remembered and we set this to be equal to the memory size, M. Since our main argument is that of easy adaptivity, we believe that it is an important goal to achieve good adaptive behavior with low tuning effort. We later contrast this to the multitude of parameters for the SEQ and EELRU algorithms.

4.2 Simulation Results: Adaptive vs. Component Policies

Our first results compare the performance of our adaptive policy relative to the policies it adapts over. Specifically, Figures 1 and 2 show the behavior (faults incurred) of our

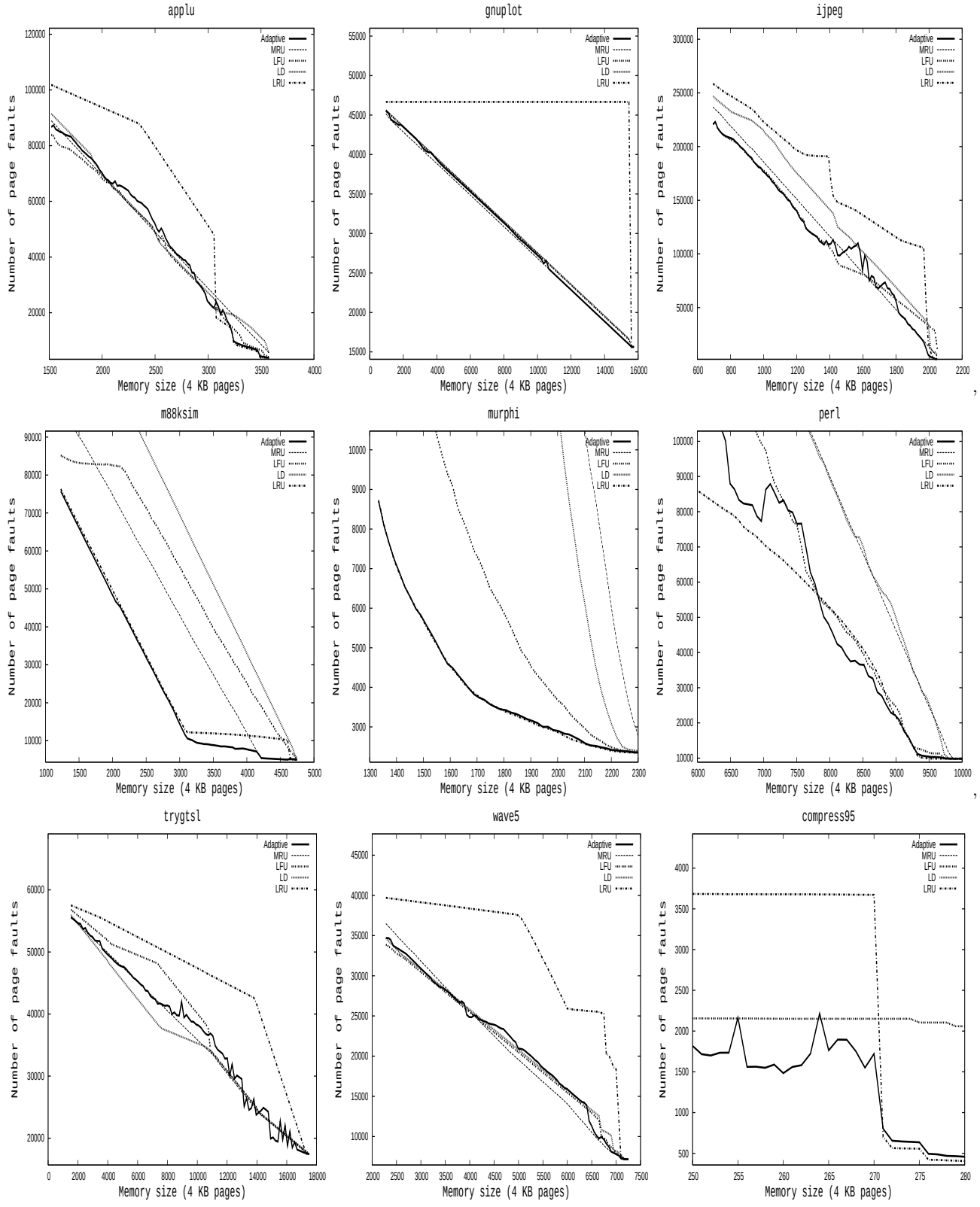

Figure 1: Comparison of adaptive scheme with the policies it adapts over for the 8 traces from the SEQ study and the trace for compress95.

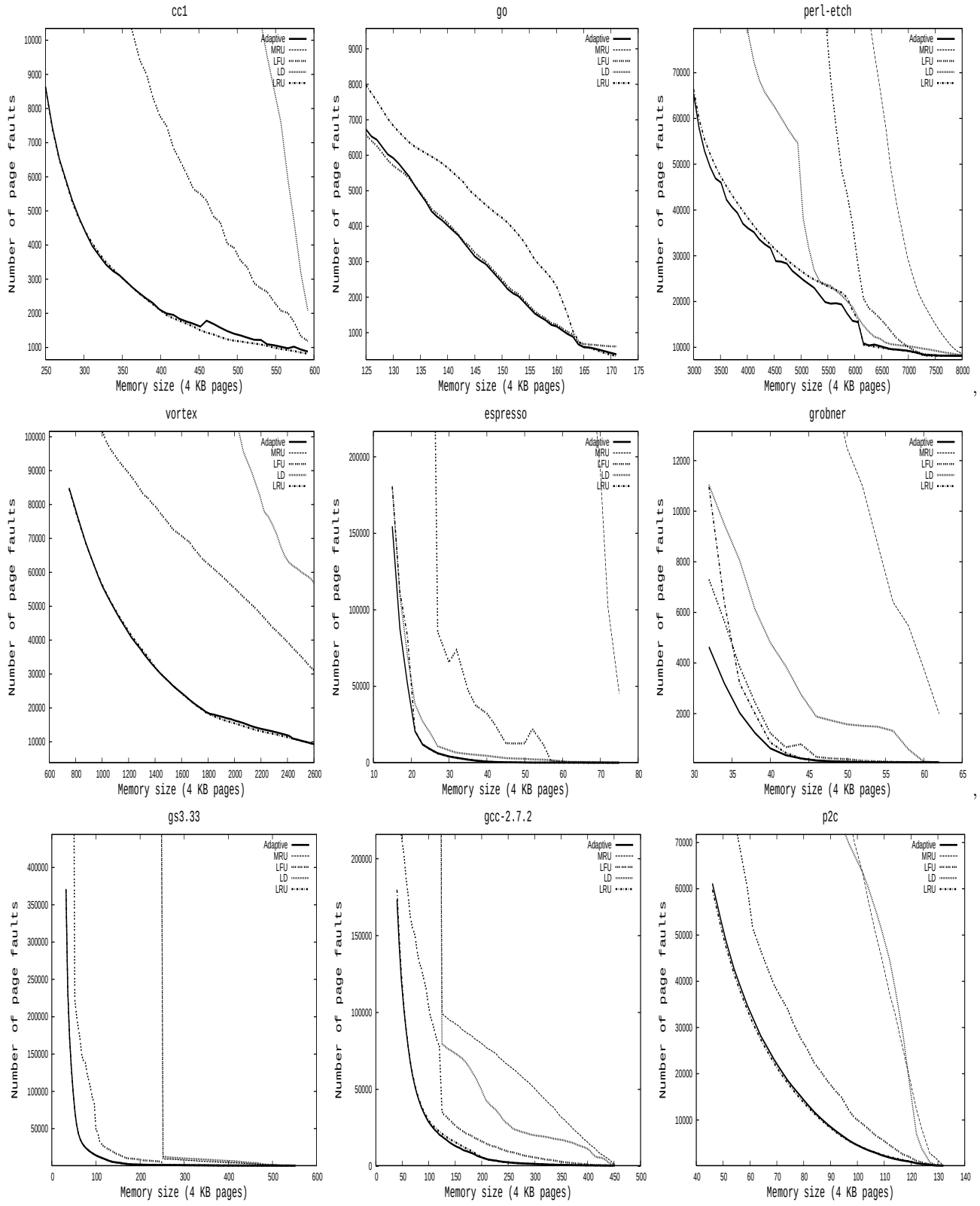

Figure 2: Comparison of adaptive scheme with the policies it adapts over for the 4 SPEC95 traces and the 5 small-scale traces.

adaptive policy compared to LRU, "MRU" (the most aggressive of the 5 MRU-like policies we use), "LFU", and our loop-detection (LD) policy. Results for the off-line optimal replacement algorithm, OPT, on the same programs are shown in later figures to keep the plots from becoming too cluttered. (Despite our best efforts, the plots require some effort to follow, due to the amount of information they contain, but the Adaptive policy (solid line) and the LRU policy should be easy to distinguish from the rest.) We show the results for the 8 traces from the SEQ study, the 5 SPEC95 Etch traces, and 5 of the small-scale traces (we omit lindsay, which is the least interesting and our policy performs identically to LRU for it). The curves for MRU and LD are sometimes not visible in the plots since their values exceed the shown range. (These algorithms would never be used alone in a realistic setting, as they are easily fooled to produce very bad results. Therefore we do not adjust the scale of our figures based on them.)

The first observation about these figures is that LFU, LD, and MRU seem to often perform better than LRU for the simulated workloads (although they also often perform catastrophically). One reason for this counterintuitive result is that, as explained earlier, our MRU, LD, and LFU are really compositions of the above algorithms with an LRU policy: the m most recently accessed blocks are guaranteed to be in memory, to ensure that the simulation is valid (values of m vary per trace, but typically reach up to 30% of the smallest simulated memory for the trace). Hence, much of the benefit of LRU in exploiting temporal locality is already obtained for the "LFU", "LD", and "MRU" versions plotted in the figures. Another reason that these algorithms often perform well is that several of the tested programs (e.g., gnuplot or trygtsl) have large linear loops, for which LRU performs pessimally.

As can be seen in Figures 1 and 2, our Adaptive policy is almost always the bottom-most curve (or very close to it), incurring the fewest faults. Specifically, compared to LRU, the adaptive policy performs noticeably worse only for the perl trace and then only for small memories. In most other cases, the Adaptive curve is far below the LRU curve by significant amounts (e.g., for applu, gnuplot, ijpeg, m88ksim and large memories, perl and medium-range memories, trygtsl, wave5, compress95, go, perl-etch, and grobner). Compared to MRU, LFU, and LD, our adaptive policy is clearly far superior—these policies are easily tricked into extremely bad behavior (e.g., for m88ksim, murphi, perl, cc1, perl-etch, vortex, and all of the small-scale traces). In a few cases, our adaptive policy is not just imitating the best one of the other policies, but is better than all of them (e.g., see the end range of the m88ksim plot, the medium range of the perl plot, the compress95 plot, the perl-etch plot, and the grobner plot). This suggests that the application changes behavior during its runtime so that different replacement policies are best for different execution phases, and that our policy adapts successfully to the phase behavior changes.

4.3 Simulation Results: Adaptive vs. EELRU, SEQ

Our next comparison is between our adaptive policy and ad hoc adaptive policies like SEQ and EELRU. Recall that our adaptivity scheme has very low tuning requirements. Additionally, our adaptivity is much simpler conceptually than either SEQ or EELRU, without being heavier-weight in terms of implementation. Both EELRU and SEQ are hand-optimized algorithms with many tuning parameters and non-negligible space or time complexities. Specifically, SEQ tracks up to 200 sequences of faults to consecutive blocks, managed as an LRU queue. Then the algorithm picks to evict the H-th page from the head of the sequence with the most recent F-th fault, provided it has length more than L. Parameters H, F and L affect the behavior of the algorithm significantly and require careful tuning for good performance. The SEQ study [6] has a section discussing how the values of these parameters affect the performance of the traces under study, based on low-level characteristics of these traces (like the number of linear sequences in a trace and their length).

Similarly, EELRU tracks accesses not only to blocks in memory, but also blocks recently evicted. In the experiments of the latest EELRU study [18], the EELRU data structures had entries for 2.5 times as many blocks as the memory size. EELRU then picks among many possible eviction decisions (40 in the cited study) by maintaining histograms of recent references in equal-length segments of the recency space. (This is analogous to having our adaptivity scheme adapt among 40 different MRU-LRU combinations! Indeed, one can derive EELRU from our general adaptivity scheme, specialized for LRU and 40 different MRU-like policies and optimized for space using the suggestions of Section 2.4.) Other tunable parameters include settings that determine how quickly the EELRU statistics get "decayed", as well as a setting that affects how the "virtual time" (used to distinguish recent references from older ones) is affected by references to not-recently-accessed pages. All of these parameters need to be carefully tuned to obtain the results of the EELRU study. In contrast, our adaptation scheme is very simple, its important events are only faults for one of the adapted-over policies, and the "decaying" of data is done simply by remembering the last k faults.

Despite its simplicity, our adaptation scheme performs very competitively to hand-tuned policies like EELRU and SEQ. Figures 3 and 4 plot the faults incurred by each policy for the traces under study. (Since we have not re-implemented the SEQ policy, we only have results for its performance on the 8 traces from the SEQ study. The SEQ implementation is tied to the trace format used in that study and running the code on different traces appears quite complex.) The figures also show the faults incurred by the off-line optimal replacement policy (OPT) for comparison purposes. As can be seen in the figures, our Adaptive scheme performs better than SEQ for four traces (applu, ijpeg, murphi, perl) and only slightly worse (and not for all memory ranges) for m88ksim, trygtsl, and wave5. Similarly, our Adaptive policy is quite competitive to EELRU, mostly outperforming it for applu, gnuplot, ijpeg, trygtsl, and go, and being outperformed for perl, compress95, cc1 (only briefly), and perl-etch. (Note that the differences in behavior for the compress95 plot seem abrupt but this is largely due to the low absolute numbers of faults for all policies involved and to the normalization of the scale. Compare to Figure 1 to see the same behavior in a plot that also includes the much higher LRU fault values.)

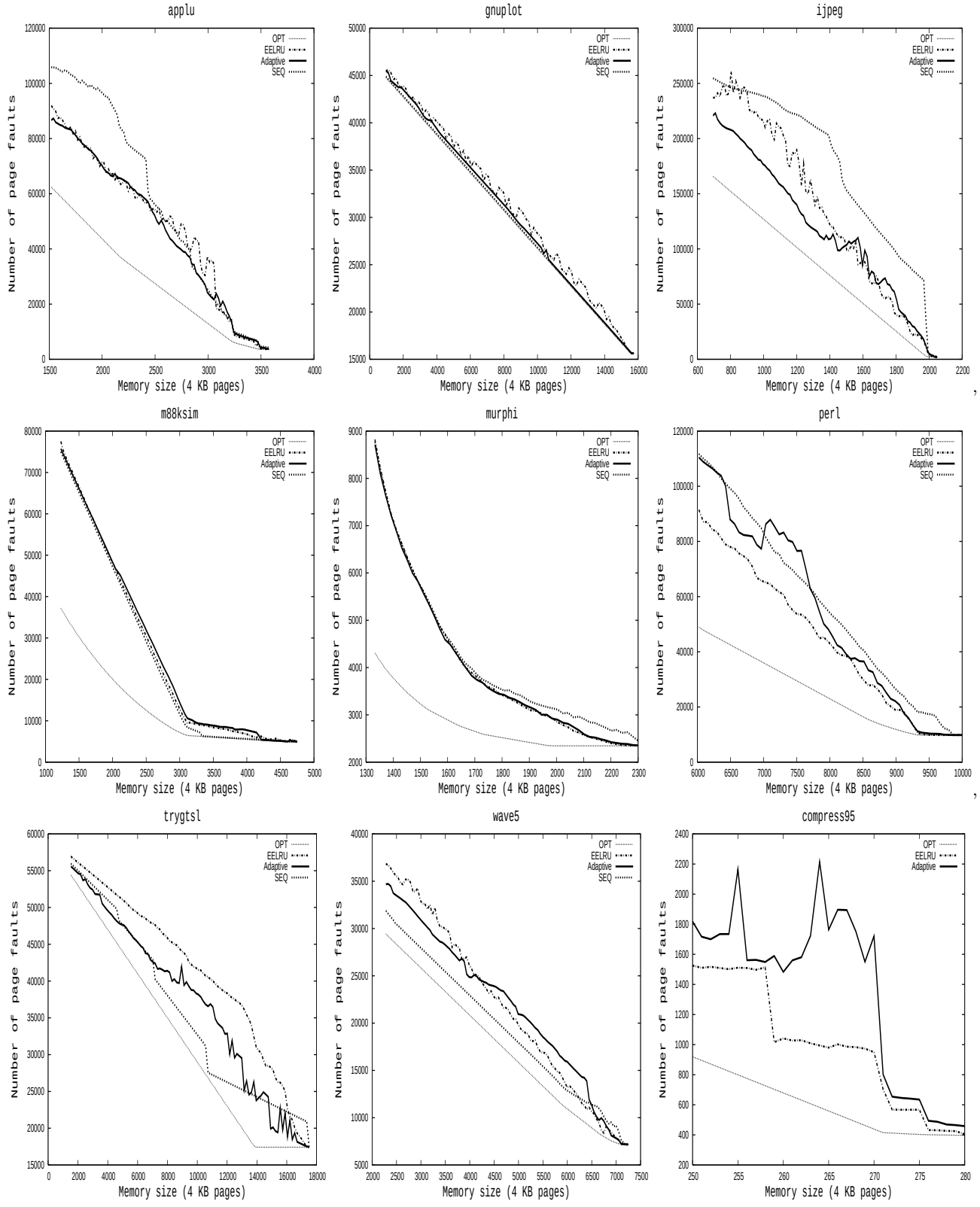

Figure 3: Comparison of adaptive scheme with SEQ and EELRU for the 8 traces from the SEQ study and the trace for compress95. SEQ is plotted for the first 8 traces only.

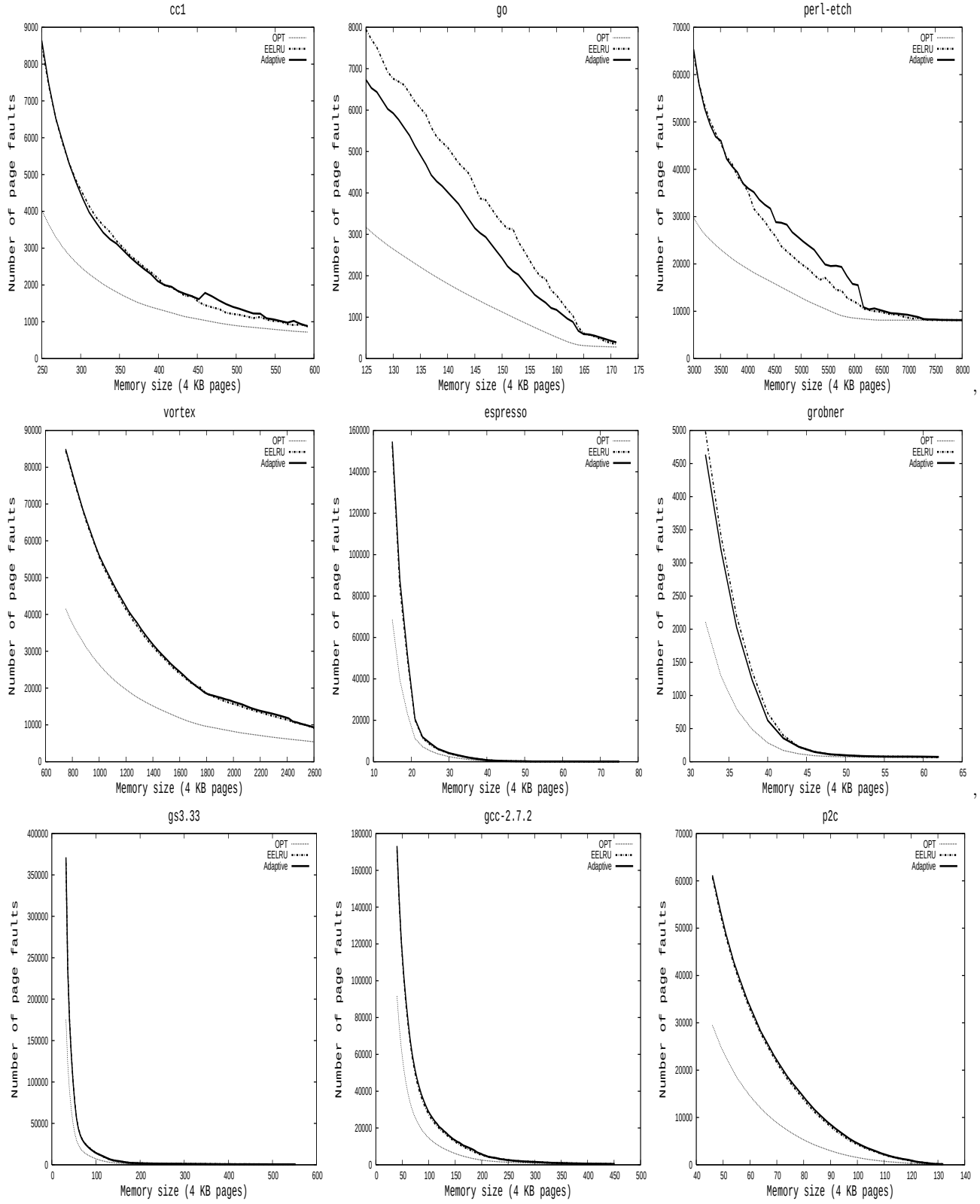

Figure 4: Comparison of adaptive scheme with EELRU for the 4 SPEC95 traces and the 5 small-scale traces.

In summary, the performance of our adaptivity scheme is remarkable, given its simplicity, generality, and lack of need for tuning. The results shown were obtained by just putting together what seemed like a reasonable set of policies to adapt over and applying our general adaptation scheme to them with virtually no tuning.

5. CONCLUSIONS

In this paper, we presented the idea of adaptive replacement policies using a very general adaptation scheme, and supported this idea with theoretical results and experimental measurements. There are many possibilities for improvement of our basic adaptation scheme, and we hope that they can inspire other researchers to continue along this direction. Although several open problems remain (e.g., can we prove low robustness bounds for an algorithm that adapts among multiple policies and not just two at a time?) we believe that our work shows convincingly that general adaptivity is easy and that it has significant advantages at the policy level. At the algorithm level, our approach is also usually quite feasible, although its time and space complexity depends on the policies being adapted over. In practical terms, our results show that it is possible to address the shortcomings of any specific replacement policy, by composing it with others that behave well for the desired scenarios. Thus, we hope that the ideas presented here will prove interesting in the design of future caching mechanisms.

6. ACKNOWLEDGMENTS

This research was supported by the NSF through grants CCR-0238289 and CCR-0220248, and by the Georgia Electronic Design Center.

7. REFERENCES

[1] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, Jan. 1971.

[2] M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth. Paging studies made on the I.C.T. ATLAS computer. *Information Processing, IFIP Congress Booklet D*, 1968.

[3] J. Choi, S. H. Noh, S. L. Min, E.-Y. Ha, and Y. Cho. Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme. *IEEE Transactions on Computers*, 51(7):793–800, July 2002.

[4] P. J. Courtois and H. Vantilborgh. A decompsable model of program paging behavior. *Acta Informatica*, 6:251–275, 1976.

[5] M. A. Franklin and R. K. Gupta. Computation of pf probabilities from program transition diagrams. *Communications of the ACM*, 17:186–191, 1974.

[6] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, pages 115–126, 1997.

[7] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, 2002.

[8] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, pages 47–58, 1999.

[9] A. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. In *IEEE Symposium on the Foundations of Computer Science*, pages 208–217. IEEE Computer Society Press, 1992.

[10] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, pages 134–143, 1999.

[11] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, Dec. 2001.

[12] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows NT. In *25th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, 1998.

[13] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX File and Storage Technologies (FAST)*, Mar. 2003.

[14] H. Muramatsu and H. Negishi. Page replacement algorithm for large-array manipulation. *Software Practice and Experience*, 10:575–587, 1980.

[15] J. Robertson and M. Devarakonda. Data cache management using frequency-based replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, 1990.

[16] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[17] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and efficient adaptive page replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, pages 122–133, 1999.

[18] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, July 2003.

[19] J. R. Spirn. Distance string models for program behavior. *IEEE Computer*, 9:14–20, 1976.

[20] R. Turner and H. Levy. Segmented FIFO page replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of computer systems*, 1981.

8. APPENDIX: PROOF OF ROBUSTNESS OF $AB(K)$

We first need a lemma analogous to Lemma 1.

LEMMA 2. *Consider the same reference sequence processed by replacement algorithms A, B and $AB(k)$. If at a certain point a block is both in the buffer managed by A and in the buffer managed by B then it is also in the buffer managed by $AB(k)$.*

Proof: Analogous to the proof of Lemma 1.□

For our main theorem we assume that $k \leq M$. (If $k > M$ the robustness result needs to be adjusted by an additive factor or the first $k - M$ replacements need to be handled specially in the definition of $AB(k)$.)

THEOREM 2. *Adaptive replacement algorithm $AB(k)$, for $k \leq M$, is robust with respect to both A and B. Specifically, $AB(k)$, as defined above, never incurs more than 3 times as many faults as either A or B.*

Proof: Define a function f:

$$f(i) = \begin{cases} 1 & \text{if the } i\text{-th most recent fault for either } A \text{ or } B \\ & \text{was a fault for } A \\ 0 & \text{otherwise} \end{cases}$$

If fewer than i faults have occurred so far, $f(i) = 0$. Note also if the i-th most recent fault for either A or B was a fault for *both* A and B, then $f(i) = 1$ per the above definition. (This choice turns out to not affect our results.) Then the number s computed in the course of execution of $AB(k)$ (i.e., the number of faults for algorithm A among the k latest faults for either A or B) is just $\sum_{i=1}^{k} f(i)$.

We define three "potential" quantities and examine how their values change on every fault for either algorithm A or B.

Let $p = \sum_{i=1}^{k}(k+1-i)f(i)$. The p quantity is a measure of the potential for deviation from algorithm A such that more recent data are favored linearly more.

Let d_A be the number of blocks currently in $AB(k)$'s buffer that are not in A's buffer at the same point in the execution.

Let d_B be the number of blocks currently in $AB(k)$'s buffer that are not in B's buffer at the same point in the execution.

We observe that $0 \leq p \leq \frac{k(k+1)}{2}$ and $0 \leq s \leq k$. The above values of p, d_A, and d_B change as follows on every fault (we denote the new values p', d'_A, and d'_B):

(Note that according to Lemma 2 a hit for both A and B implies a hit for $AB(k)$ and none of the three potentials changes.)

1. if $s > \frac{k}{2}$
   (a) fault for B, hit for A, hit for $AB(k)$: $p' = p - s$, $d'_A = d_A$, $d'_B \leq d_B$
   (b) fault for B, hit for A, fault for $AB(k)$: $p' = p - s$, $d'_A \leq d_A$, $d'_B \leq d_B$
   (c) fault for B, fault for A, fault for $AB(k)$: $p' = p + k - s$, $d'_A \leq d_A + 1$, $d'_B \leq d_B$
   (d) fault for B, fault for A, hit for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A$, $d'_B \leq d_B$
   (e) hit for B, fault for A, hit for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A$, $d'_B = d_B$
   (f) hit for B, fault for A, fault for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A + 1$, $d'_B < d_B$
2. if $s \leq \frac{k}{2}$
   (a) fault for B, hit for A, hit for $AB(k)$: $p' = p - s$, $d'_A = d_A$, $d'_B \leq d_B$
   (b) fault for B, hit for A, fault for $AB(k)$: $p' = p - s$, $d'_A < d_A$, $d'_B \leq d_B + 1$
   (c) fault for B, fault for A, fault for $AB(k)$: $p' = p + k - s$, $d'_A \leq d_A$, $d'_B \leq d_B + 1$
   (d) fault for B, fault for A, hit for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A$, $d'_B \leq d_B$
   (e) hit for B, fault for A, hit for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A$, $d'_B = d_B$
   (f) hit for B, fault for A, fault for $AB(k)$: $p' = p+k-s$, $d'_A \leq d_A$, $d'_B \leq d_B$

We do not show all the case-by-case reasoning needed to derive the above values. As a single example, consider, for instance, case 1.c. In this case, there is a fault for A. Thus the previous k faults for either A or B become "older" and the quantity p is reduced by s (what was before $f(i)$ is now $f(i+1)$ and p is reduced by s: the total of non-zero $f(i)$s in the k most recent faults). At the same time, the most recent fault adds k to p ($f(1)$ is 1). Thus, in total, $p' = p + k - s$. Since the fault is both for A and for $AB(k)$, one extra common page will exist in both buffers after this reference is processed. At the same time, however, the $AB(k)$ algorithm is in B-mode (because $s > \frac{k}{2}$), thus A and $AB(k)$ may pick two different pages to evict and these pages could have been common to both their buffers before. Thus d_A (the number of pages in $AB(k)$'s buffer that are not in A's buffer) may increase but at most by one (hence the $d'_A \leq d_A + 1$). With similar reasoning, d_B may decrease or stay the same.

Now we are ready to show the first part of our result. Consider the total potential function $t_A = p + d_A\frac{k}{2}$. We have that $t_A \geq 0$. By t'_A we denote the new value of t_A after a fault for either A or B. If $AB(k)$ suffers a fault that is not a fault for A (cases 1.b, 2.b) we have that $t'_A < t_A - \frac{k}{2}$ (either because p is reduced by s and $s > \frac{k}{2}$ or because d_A is reduced by one). t_A can only grow in cases 1.c, 1.d, 1.e, 1.f, 2.c, 2.d, 2.e, and 2.f. In each of them, A suffers a fault and $t'_A \leq t_A + k$. Originally, $t_A = 0$. Therefore, for each fault of $AB(k)$ that is not a fault of A the t_A quantity decreases by at least $\frac{k}{2}$. It can only increase by at most k if A incurs a fault and it has to stay non-negative. Hence, $AB(k)$ can suffer at most three times as many faults as A.

Similarly, define the total potential function $t_B = \frac{k(k+1)}{2} - p + d_B\frac{k}{2}$. We have that $t_B \geq 0$. By t'_B we denote the new

value of t_B after a fault for either A or B. If $AB(k)$ suffers a fault that is not a fault for B (cases 1.f, 2.f) we have that $t'_B \leq t_B - \frac{k}{2}$. t_B can only grow in cases 1.a, 1.b, 2.a, and 2.b. In each of them, B suffers a fault and $t'_B \leq t_B + k$. Since $k \leq M$, by the time the memory becomes full (i.e., at the initial point when M distinct pages have been referenced) $d_B = 0$ (because no eviction has taken place) and $p = \frac{k(k+1)}{2}$ (because all initial faults are also faults for algorithm A). Thus, at this point $t_B = 0$. By this point, all three algorithms have incurred exactly M faults. From then on, for each fault of $AB(k)$ that is not a fault of B the t_B quantity decreases by at least $\frac{k}{2}$. It can only increase by k if B incurs a fault and it has to stay non-negative. Hence, $AB(k)$ can suffer at most three times as many faults as B. □