Partitioning Techniques for Partially Protected Caches in Resource-Constrained Embedded Systems

KYOUNGWOO LEE
University of California, Irvine
AVIRAL SHRIVASTAVA
Arizona State University
and
NIKIL DUTT and NALINI VENKATASUBRAMANIAN
University of California, Irvine

Increasing exponentially with technology scaling, the soft error rate even in earth-bound embedded systems manufactured in deep subnanometer technology is projected to become a serious design consideration. Partially protected cache (PPC) is a promising microarchitectural feature to mitigate failures due to soft errors in power, performance, and cost sensitive embedded processors. A processor with PPC maintains two caches, one protected and the other unprotected, both at the same level of memory hierarchy. The intuition behind PPCs is that not all data in the application is equally prone to soft errors. By finding and mapping the data that is more prone to soft errors to the protected cache, and error-resilient data to the unprotected cache, failures induced by soft errors can be significantly reduced at a minimal power and performance penalty. Consequently, the effectiveness of PPCs critically hinges on the compiler's ability to partition application data into error-prone and error-resilient data. The effectiveness of PPCs has previously been demonstrated on multimedia applications—where an obvious partitioning of data exists, the multimedia data is inherently resilient to soft errors, and the rest of the data and the entire code is assumed to be

This is an expanded version of a conference paper published in *Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2008)*. The current article extends the previous paper by: (1) exploring the design space for the instruction PPC (partially protected caches) in Section 7.3, (2) Proposing two more heuristic algorithms to efficiently find the best partitions with minimum vulnerability under the performance penalty in Section 5.2; (3) presenting extensive experiments from Section 7 with the comprehensive energy consumption model from Section 6; and (4) differentiating our work from comprehensive related work in Section 2 and preliminary experiments in Section 4.

Authors' addresses: K. Lee, N. Dutt, and N. Venkatasubramanian, Department of Computer Science, University of California, Irvine, CA 92697; K. Lee, email: kyoungwl@ics.uci.edu; A. Shrivastava, Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281.

DOI 10.1145/1835420.1835423 http://doi.acm.org/10.1145/1835420.1835423

error-prone. Since the amount of multimedia data is a quite significant component of the entire application data, this obvious partitioning is quite effective. However, no such obvious data and code partitioning exists for general applications. This severely restricts the applicability of PPCs to data caches and instruction caches in general. This article investigates vulnerability-based partitioning schemes that are applicable to applications in general and effectively reduce failures due to soft errors at minimal power and performance overheads.

Our experimental results on an HP iPAQ-like processor enhanced with PPC architecture, running benchmarks from the MiBench suite demonstrate that our partitioning heuristic efficiently finds page partitions for data PPCs that can reduce the failure rate by 48% at only 2% performance and 7% energy overhead, and finds page partitions for instruction PPCs that reduce the failure rate by 50% at only 2% performance and 8% energy overhead, on average.

Categories and Subject Descriptors: B.3.4 [**Memory Structures**]: Reliability, Testing, and Fault-Tolerance

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Page partitioning technique, partially protected cache, soft error, vulnerability, embedded systems

ACM Reference Format:

Lee, K., Shrivastava, A., Dutt, N., and Venkatasubramanian, N. 2010. Partitioning techniques for partially protected caches in resource-constrained embedded systems. ACM Trans. Des. Autom. Electron. Syst. 15, 4, Article 30 (September 2010), 30 pages.
DOI = 10.1145/1835420.1835423 http://doi.acm.org/10.1145/1835420.1835423

1. INTRODUCTION

System reliability is becoming the paramount concern in system design in the deep submicron era [ITRS 2005]. Four decades of technology scaling has brought us to a point where the transistors have become extremely susceptible to even small fluctuations in voltage levels, slight noise in the power supply, signal interference, and even cosmic particle strikes [Hazucha and Svensson 2000; Wrobel et al. 2001; Shivakumar et al. 2002; Baumann 2005]. All of these effects can temporarily toggle the logic value of a transistor, so it is therefore called a transient fault. Such transient faults are random and nondestructive, that is, resetting the device restores normal behavior. Previous investigation finds cosmic radiation strikes to be responsible for more transient faults than all the other reasons combined [Baumann 2005]. A high energy radiation particle, for example, an alpha particle, a neutron, or a free proton, may strike the diffusion region of a CMOS transistor and produce charge that can result in toggling the logic value of the gates or flip-flops. This phenomenon of change in the logic state of a transistor is called an *upset*. An upset may have catastrophic consequences including the application generating incorrect results, accessing unauthorized memory regions, crashing, or going into an infinite loop. The incorrect or erroneous behavior of an application due to upsets is called a *failure*.

Not all upsets result in failures, and upsets can be masked due to effects such as electrical masking (upset is not strong enough to reach the next latching element), logical masking (upset on the input of the gate does not affect its output), latching-window masking (upset does not reach latch at the latching time), microarchitectural masking (upset happens on a variable that is no longer used), and software masking (upset happens in a function, whose output is not used) [Shivakumar et al. 2002]. An upset will become a failure if it is not masked by any of these effects.

Owing to the effectiveness of latching-window masking, upsets in memory elements have significantly higher probability of causing a failure than upsets in combinational logic [Gaisler 1997; Liden et al. 1994]. In addition, since memory elements may occupy the majority of the chip area, and the fact that they operate on lower voltages than combinational circuits, they are extremely vulnerable to radiation, and radiation-induced faults. In fact, according to Mitra et al. [2005], more than 50% of soft errors occur in memories.

The use of error detection code (EDC) like a parity check has been suggested to protect the caches from soft errors. However, a parity check only detects an error. While it is possible to employ simple error correction code (ECC)-based techniques in off-chip and lower levels of memories, such solutions are not suitable for caches, as they are highly sensitive to the performance and power overheads of redundancy-based techniques. For example, using single-bit error correction and double-bit error detection (SEC-DED) codes may increase the cache access time by 95% [Li and Huang 2005], power consumption by 22% [Phelan 2003], and area cost by 25% [Krueger et al. 2008]. While it may be possible to hide the performance penalty, it is not possible to hide the power penalty. Consequently, novel techniques are required for caches that can eventually reduce failure rates while incurring minimal power and performance overheads.

We proposed partially protected cache (PPC) architectures to mitigate failures due to of soft errors on caches at minimal power, performance, and area overheads [Lee et al. 2006, 2009]. A PPC architecture has two caches, one protected against soft errors, and the other unprotected, at the same level of memory hierarchy. The intuition behind PPC is that not all data is equally prone to soft errors, and that most of the soft errors show up in a relatively small amount of application data. Thus by protecting a small amount of data by a small protected cache, programs can be made robust at minimal power, performance, and area penalty.

Clearly the effectiveness of PPC architectures is critically hinged on the compiler's ability to partition the application data and code into error-prone and error-resilient data and code. At the microarchitecture level, we use vulnerability as the measure of how error-prone a variable is [Mukherjee et al. 2003; Asadi et al. 2005]. Vulnerability of a program variable is essentially the probability that the occurrence of a soft error in the variable in the cache will affect the program state, possibly causing a program failure. Estimating the vulnerability of a program variable is an inherently difficult task, as several program and microarchitectural factors may have a significant effect on the vulnerability of the program variable. These include, (1) the access pattern of the variable, for example, a variable that is not read by the processor, and will be overwritten, is not vulnerable, (2) the access pattern of the other program variables, for example, if the other data evicts this variable from the cache, then it will be in memory, and will not be vulnerable, and (3) cache characteristics, for example, the size, associativity, write, and allocate policies of the cache can significantly affect the time a datum is vulnerable in the cache.

Previously, PPCs have been shown to be extremely cost effective in soft error protection for multimedia applications. An obvious partitioning of data into soft error-prone and soft error-resilient data exists in multimedia applications, wherein the multimedia data itself is quite soft error-resilient. For example, in an image or video processing application, a soft error in the image or video data itself only causes a slight degradation in the quality of service (QoS). In contrast, most other data, for example, loop control variables, stack pointers, and so on, are not error-resilient. This obvious partitioning is very effective since the size of the multimedia data is quite significant.

However, no obvious data partitioning exists for general applications. To use PPC architectures for applications in general, a data partitioning scheme that divides application data and code into soft error prone and soft error resilient is extremely critical. In this article, we examine application profile to partition the application data and code into error-prone and error-resilient, enabling the use of PPC architectures for several application-specific embedded systems.

We find that Monte Carlo exploration is unable to find interesting data partitions. While Genetic Algorithm can efficiently search the exploration space, it does not achieve high reduction in vulnerability. Our partitioning heuristics is aware of runtime and vulnerability, and is therefore able to efficiently prune the search space and uncover Pareto-optimal partitions. We propose data partitioning schemes for both data PPC and instruction PPC. Experimental results on the HP iPAQ h4600-like processor memory subsystem [Hewlett Packard] running benchmarks from the MiBench suite [Guthaus et al. 2001] demonstrate that data PPC architectures can reduce the vulnerability by 48% with 2% performance and 7% energy penalty on average. In addition, instruction PPC architectures can reduce the program vulnerability by 50% while incurring 2% performance and 8% energy consumption overheads, on average.

2. RELATED WORK

Radiation-induced soft errors have been under investigation since late 1970s. Due to incessant technology scaling, soft error rate (SER) has exponentially increased [Hazucha and Svensson 2000], and now it has reached a point, where it becomes a real threat to system reliability.

2.1 Packaging Solutions

Radioactive substances such as alpha particles emitted by packaging and wafer processing materials are one of the major sources of radiation that causes soft errors in semiconductors. Thus, advances in process technology such as purification of packaging materials, radiation hardening, and elimination of Boron-10 (B^{10}) impurities, are expected to mitigate the soft errors [Baze et al. 2000]. However, the effects of interactions between high energy cosmic particles (e.g., neutrons) and radioactive materials, cannot be completely prevented [Mastipuram and Wee 2004].

2.2 Process Technology Solutions

Process technology solutions such as SOI (silicon on-insulator) processes [Musseau 1996; Roche et al. 2003] have been proposed. In order to mitigate the soft errors, they extend the depletion region or raise the capacitance, which increases the critical charge of semiconducting devices. The critical charge is the least charge to be able to invert the bit value of the memory cell. However, process engineering technology may require the cost of additional process complexity, the loss of manufacturability, and extra substrate cost [Baumann 2005].

2.3 Microarchitectural Solutions

Microarchitectural solutions attempt to reduce the number of upsets that translate into errors and/or errors that result in failures. Solutions at the microarchitecture level can be categorized based on the components where they are applied: the combinational components, the sequential components, and the memory components.

Solutions for Combinational Logic. Logic elements were considered more robust against soft errors than memory elements, mainly due to the masking effects. However, many researchers predict that the logic soft errors will become one of main contributions to system unreliability [Shivakumar et al. 2002; Baumann 2005; Nieuwland et al. 2006]. The simplest and most effective way to reduce failures due to soft errors in combinational logic is triple modular redundancy (TMR) [Pradhan 1996], which typically uses three functionally equivalent replicas of a logic circuit and a majority voter. But the overheads of hardware and power for conventional TMR exceed 200% [Nieuwland et al. 2006]. Duplex redundancy [Mohanram and Touba 2003; Nieuwland et al. 2006] is also available but it requires more than 100% area and power overheads without any optimization techniques. In order to reduce the high overheads in conventional redundancy techniques, Mohanram and Touba [2003] presented a partial error masking by duplicating the most sensitive and critical nodes in a logic circuit based on the asymmetric susceptibility of nodes to soft errors. Nieuwland et al. [2006] proposed a structural approach analyzing the soft error rate sensitivity of combinational logic to identify the critical components at circuits.

Solutions for Sequential Logic. Temporal redundancy is another main approach that has been used to combat soft errors in circuits. In order to detect soft errors, Nicolaidis [1999] applied fine time-grain redundancy within the clock cycle, greater than the duration of transient faults, by using the temporal nature of soft errors. Similarly, Anghel and Nicolaidis [2000] exploited the temporal nature to detect timing errors and soft errors by means of time redundancy. Krishnamohan and Mahapatra [2004] proposed time redundancy methodology by using the timing slack available in the propagation path from the input to the output in CMOS circuits. A razor flip-flop was presented in Ernst et al. [2003] to detect transient errors by sampling pipeline stage values with a fast clock and with a time-borrowing delayed clock.

Solutions for Memories. By far, reducing soft errors in memories has been the most extensive research topic. Error detection and correction codes (EDC and ECC) have been widely investigated and implemented as the most effective scheme to detect and correct soft errors in memory systems. However, an ECC system consists of an encoding block as well as a decoding block responsible for detection and correction, and of extra bits that store parity values. Thus, ECC-based techniques consume extra energy and incur performance delay as well as additional area cost [Pradhan 1996; Phelan 2003; Li and Huang 2005; Krueger et al. 2008], and are therefore not suitable for caches. Thus, only a few processors, such as the Intel Itanium processor [Quach 2000] protect L2 and L3 caches with ECC [Stackhouse et al. 2008], but we are not aware of any processor employing an ECC-based protection mechanism on L1 cache in resource-constrained embedded processors. This is mainly due to high overheads of ECC implementation [Kim 2006; Mohr and Clark 2006; Zorian et al. 2005]. Zhang et al. [2003] proposed in-cache replication where the dead cache block space is recycled to hold replicas of the active cache block. Also, Zhang [2005b] presented replication cache where a small fully associative cache is added to keep the replica of every write to the L1 data cache. However, these techniques incur overheads to maintain replicas. A cache scrubbing technique [Mukherjee et al. 2004] has been proposed, which can fix all single-bit errors periodically and prevent potential double-bit errors. Li et al. [2004] evaluated the drowsy cache and the decay cache, exploiting voltage scaling and shutdown schemes, respectively, in order to efficiently decrease the power leakage. They also proposed an adaptive error correcting scheme to different cache data blocks, which can save energy consumption by protecting clean data less than dirty data blocks. Kim [2006] proposed the combined approach of parity and ECC codes to generate the reliable cache system in an area-efficient way. However, they all exploit expensive error correcting codes in order to unnecessarily protect all the data.

Recently, Sugihara et al. [2007] proposed reliable cache architectures (RCA), in which the cache methods are configured to control reliability and performance, and they presented a task scheduling method to dynamically switch these operation modes between the performance and reliability in cache architectures of multiprocessor systems. Sugihara also presented task schedulding for heterogeneous multiprocessor systems, and demonstrated that a heterogeneous multiprocessor is more reliable than a homogeneous one in terms of soft error vulnerability [Sugihara 2008]. However, their RCAs are configured to switch operation modes at a task level among multiple tasks in multiprocessor systems while our partitioning techniques partition data and instructions into physically separated multiple caches within a task at a page level. Further, their constraint is a real-time requirement, that is, their task scheduling is satisfactory with 1.6 and 3.0 times longer runtimes than a conventional cache [Sugihara et al. 2007], while ours considers the minimal performance overhead, that is, only 5% runtime penalty is allowed.

Partially Protected Cache Architecture. We proposed PPC architecture and demonstrated its effectiveness in reducing the failure rate with minimal power and performance overheads [Lee et al. 2006; 2009]. However, the effectiveness of PPCs has been limited to only multimedia applications; there is no known approach to use PPCs for both data and instruction caches in general applications.

2.4 Software Solutions

Software-only techniques have been studied to protect data and code from soft errors. Both software and hardware techniques have their own advantages and disadvantages in combating the impact of soft errors. For example, hardware techniques increase the resource cost, but with high effectiveness indetecting and even correcting errors while software solutions mostly do not incur hardware costs, but provide minimal coverage, for example only error detection.

Reis et al. [2005a] presented software-implemented fault tolerance (SWIFT) for soft error detection by exploiting unused resources and enhancing control-flow checking. Also, Lucchetti et al. [2005] proposed software mechanisms to tolerate soft errors by leveraging virtual machine and memory sharing techniques. However, they are limited to detecting errors, and must be used in conjunction with recovery techniques. Through user-specified annotations, the compiler can separate and map data elements in programs either to a reliable domain that has protection techniques against soft errors, or to an unreliable domain without protection [Chen et al. 2005]. But it requires annotation for important data by user specification.

Soft error detection in software is extremely expensive in terms of delay, while it can be done without much overhead in hardware. In contrast, since the soft error rate is very low (as compared to the processor clock cycle), soft error correction is efficient in software but incurs too much overhead in hardware. Consequently, a combined approach that achieves the best of both hardware and software solutions is very efficient. Reis et al. [2005b] proposed a software-hardware hybrid suite named CRAFT (CompileR Assisted Fault Tolerance) combining a software-only approach, SWIFT (Software Implemented Fault Tolerance), with a hardware-only approach, RMT (Redundant MultiThreading). The CRAFT approaches are promising alternatives since they can trade off performance, reliability, and hardware costs between software-only approaches and hardware-only approaches. Hu et al. [2005] proposed a hardware-software hybrid approach, which duplicates instructions directed by compilers and supported by architecture such as register and address queues. The performance and reliability of this proposal are located between no duplication and full duplication for error detection in VLIW architectures. However, both hybrid techniques are for soft error detection rather than for soft error correction.

PPC architecture with software page partitions is promising as a joint solution of hardware-software techniques for error correction in resource-constrained embedded systems. The compiler separates the failure-critical and failure-noncritical data and maps each of them into the two caches in a PPC for the selective data protection technique in multimedia applications [Lee et al. 2006; 2009]. However, there is no partitioning technique for general applications.

Our Contribution. This article investigates the software challenges in using PPCs. The contribution of this article is in developing techniques to utilize

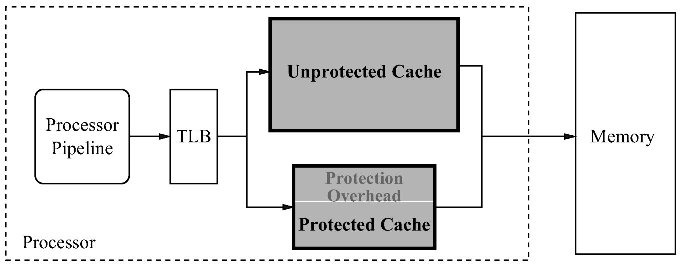


Fig. 1. Partially protected cache architecture: a protected cache and an unprotected cache at the same level of hierarchy.

PPC architectures for general applications and establishing PPC as an effective microarchitectural solution to mitigate failures due to soft errors, not only for data caches but also for instruction caches.

3. PARTIALLY PROTECTED CACHES AND PROBLEM DEFINITION

In a processor with partially protected cache (PPC), the processor has two caches at the same level of memory hierarchy. As shown in Figure 1, one of two caches is protected from soft errors, while the other is unprotected. Any soft error protection mechanism can be implemented in the protected cache, for example, increasing the thickness of the oxide layer of the transistors in the cache, or adding redundancy logic like SEC-DED (single-bit error correction and double-bit error detection). To keep the access latencies of the protected cache and the unprotected cache the same, the protected cache is typically smaller than the unprotected cache.

Each page in the memory is mapped exclusively to one of the caches in a PPC architecture. The page mapping is set as a page attribute by the compiler. The mapping of the pages present in the cache resides in the translation lookaside buffer (TLB). On a cache access, first a TLB lookup is performed to find out if the page is present in the cache, and if so, in which one. Thus, only one cache lookup is performed per cache access.

While PPC architectures can be very effective in reducing the failure rate with minimal performance and power overheads, the effectiveness hinges on the ability to partition the application data and code between the two caches. To demonstrate the need and effectiveness of page partitioning to reduce the failure rate, we performed a small experiment. First we mapped all the application pages to the unprotected cache, and then moved the pages to the protected cache one by one. Figure 2 plots the failure rate at each step of this exploration for benchmark *susan corners* from the MiBench suite on a modified *sim-outorder* simulator from SimpleScalar [Burger and Austin 1997] to model a HP-iPAQ like system. To estimate the failure rate, we injected soft errors on a randomly selected bit at data caches for each execution of the benchmark at an SER of 10^{-11} per KB per instruction for single-bit errors. Each execution is defined as a success if it ends within twice the normal execution time and returns the correct output. Otherwise, it is a failure. The failure rate is calculated as

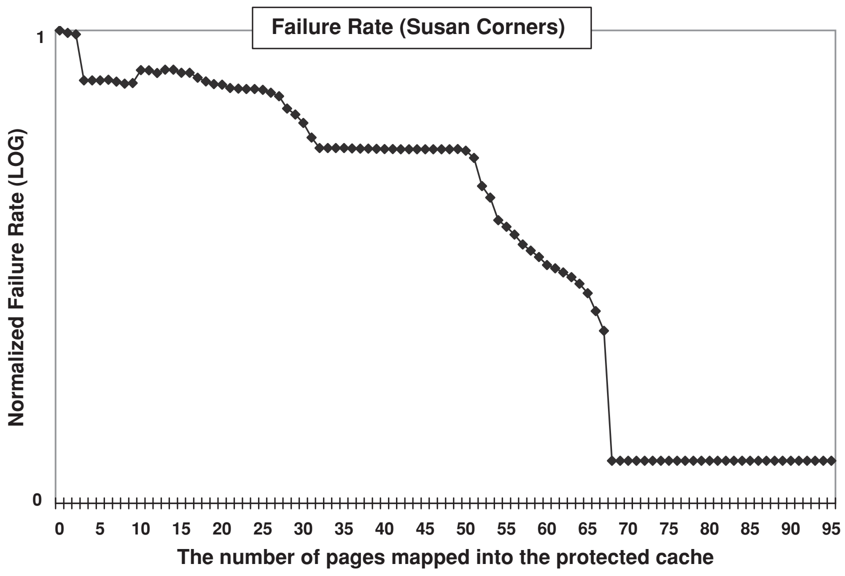


Fig. 2. Failure rate reduction by moving pages from the unprotected cache into the protected cache one by one in a PPC.

the ratio of the number of failures to the number of runs. Figure 2 shows that the failure rate of the application drops rapidly as pages are moved from the unprotected cache to the protected cache. Note that the y-axis is logarithmic and the failure rate of each mapping is normalized to the failure rate of the default mapping, where all the application pages are mapped into the unprotected cache in a PPC in Figure 2. However, the pages have to be carefully moved to the small protected cache, as it is small; mapping too many pages to the small cache may increase the cache misses and result in a significant degradation of performance and increase in the energy consumption. Indeed, the performance can decrease by up to 27% for *susan corners* when all pages are mapped to the 256 byte protected cache as compared with mapping all pages to the 4 KB unprotected cache in a PPC. So there is a definite need to study the tradeoff between the failure rate and performance (energy consumption) in finding the partitions for PPC architectures.

Therefore, the partitioning problem is a multiobjective optimization problem, in which we need to reduce the failure rate at minimal performance degradation, and minimal increase in the energy consumption. Since, even medium sized applications use a large number of data pages, our benchmarks were selected from the MiBench suite [Guthaus et al. 2001] access 27–95, on average 56 pages. Owing to their exponential complexity, enumerative techniques (e.g. trying all the possible page partitions and picking up the best one) do not work.

We formulate our problem as: Given an allowable performance degradation, determine the page partitioning to minimize the failure rate at minimal energy penalty.

```
CacheAccess (cmd, addr, nbytes, now)
01: if (CACHE_MISS)
02:     VDS.doEvent(E, evictLine.getAddr(), lineSize, now, evictLine.isDirty())
03:     now += CACHE_MISS_PENALTY
04:     VDS.doEvent(I, addr.getAddr(), lineSize, now)
05: endIf
06: if (cmd == READ)
07:     VDS.doEvent(R, addr, nbytes, now)
08: else
09:     VDS.doEvent(W, addr, nbytes, now)
10: endIf
```

Fig. 3. Modified cache access function.

4. VULNERABILITY: MICROARCHITECTURAL METRIC FOR FAILURE RATE

To efficiently choose pages to be mapped to the protected cache, we need a metric to quantitatively compare page partitions in terms of susceptibility to soft errors. Previous studies [Mukherjee et al. 2003; Asadi et al. 2005; Zhang 2005a; Wang et al. 2006] have tried to formulate such an estimate. The most closely related one is the critical time metric in Asadi et al. [2005]. However, they loose accuracy by computing the metric at a word level, and they have not demonstrated the accuracy of their approach. To partition the data into the protected and unprotected caches in a PPC, we use the metric of vulnerability, based on the critical time metric in Asadi et al. [2005], while we estimate the vulnerability at a byte level and consider more comprehensive events in the caches (e.g., an eviction event). We observe that if an error is injected into a variable that will not be used, the error does not matter. However, if the erroneous value will be used in the future, then it will result in a failure. Thus data is defined to be vulnerable for the time it is in the unprotected cache until it is eventually read by the processor or written back to the memory. If data will be overwritten, or if it will not be written back to the memory (typically because it is not dirty), then it is not vulnerable. Figure 3 shows the modifications required to the cache access function to compute the vulnerability in an execution.

We consider four cache events, *Incoming (I)*, *Read (R)*, *Write (W)*, and *Eviction (E)*. All the cache events are registered in the VDS (vulnerability data structure). In case of a cache miss, only if the evicted line is dirty (line 02 in Figure 3), an eviction event for the whole line will be registered in the VDS. The incoming line will register an incoming event (line 04), while read and write events will be registered on a cache read and a cache write, respectively (lines 07, 09).

What happens in each event is described in Figure 4. For each event, the event and the time stamp on each byte are updated (lines 03, 05, 09, 11, 15, 16, 20, 25). The variable, cv, computes the cumulative vulnerability of each byte. When a byte is read, the elapsed time from the last event to this read is added to cv (line 11). However, when a byte is written, the time from the last event to this write event is ignored.

When a byte is evicted from the cache, if the byte is dirty, it will be written back into the memory, and therefore the time from the last event to this eviction is added to cv (line 23). However, if this line is not dirty and if this line will

```
doEvent (event, addr, nbytes, now, dirty)
01: if $(event == I)$
02:     for $(i = 0; i < nbytes; i++)$
03:         $VDS[addr + i].event = I$
04:         $VDS[addr + i].cv\_le = 0$
05:         $VDS[addr + i].time = now$
06:     endFor
07: else if $(event == R)$
08:     for $(i = 0; i < nbytes; i++)$
09:         $VDS[addr + i].event = R$
10:         $VDS[addr + i].cv += now - VDS[addr + i].time$
11:         $VDS[addr + i].time = now$
12:     endFor
13: else if $(event == W)$
14:     for $(i = 0; i < nbytes; i++)$
15:         $VDS[addr + i].event = W$
16:         $VDS[addr + i].time = now$
17:     endFor
18: else if $(event == E)$
19:     for $(i = 0; i < nbytes; i++)$
20:         $VDS[addr + i].event = E$
21:         $VDS[addr + i].cv\_le = VDS[addr + i].cv$
22:         if $(dirty)$
23:             $VDS[addr + i].cv += now - VDS[addr + i].time$
24:         endIf
25:         $VDS[addr + i].time = now$
26:     endFor
27: endIf
```

Fig. 4. Update vulnerability on each event.

```
ComputeVulnerability (VDS)
01: for $(addr = 0; addr < |VDS|; addr++)$
02:     if $(VDS[addr].event == E)$
03:         $vul += VDS[addr].cv\_le$
04:     else
05:         $vul += VDS[addr].cv$
06:     endIf
07: endFor
08: return $vul$
```

Fig. 5. Compute the vulnerability of an application by adding byte vulnerabilities.

not be brought into the cache again, then the time from the last event to this eviction should not be considered (line 21). Since whether the line will be brought back into the cache or not is unknown, this value is maintained in a separate variable cv_le, which denotes the vulnerability of the byte after an eviction, assuming that it will not be brought back into the cache. If it is later brought into the cache, the variable is set to zero on a byte incoming event (line 05).

The final computation of the vulnerability of an application is computed after the end of the simulation, which is described in Figure 5. The application vulnerability is the sum of the vulnerabilities of each byte. Only if the last event on a byte was an eviction, the cv_le is added (line 03). Otherwise, cv is added (line 04).

To validate our idea of using vulnerability as a failure rate metric, we simulated the *susan corners* benchmark for various L1 cache sizes. Figure 6 plots the vulnerability and the failure rate obtained by the simulations. The failure rate

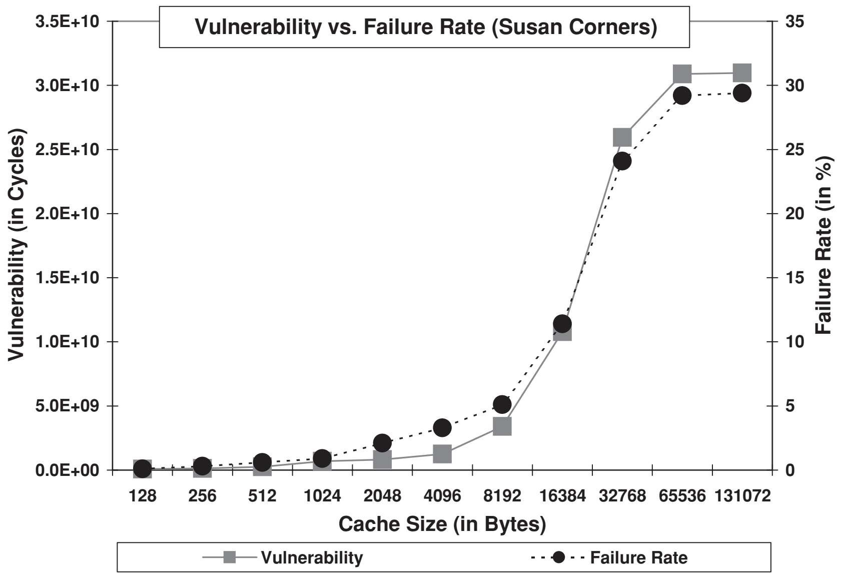


Fig. 6. Vulnerability is a good metric for estimating failure rate.

is calculated in % by multiplying by 100, the number of failures divided by the total number of runs, and the vulnerability is measured in cycles from the modified SimpleScalar *sim-outorder* simulator for the benchmark. Figure 6 shows that the shape of the vulnerability closely matches the failure rate curve. Other applications also show similar trends. On average, the error in predicting the failure rate using the vulnerability metric is less than 5%. In this article, we use vulnerability as the metric to estimate the failure rate, and perform automated design space exploration to decide the page partitioning between the two caches of a PPC. Note that the main benefit obtained from the vulnerability metric other than this design space automation is that we can save exploration time as compared to estimating the failure rate by simulation-based enumerative techniques.

Then we need a strategy to find pages causing high vulnerability of an application and to map them into the protected cache in a PPC architecture. We have developed a simple strategy, which (1) profiles a vulnerability for each page, and (2) explores the partition by moving a page with the highest vulnerability among the remaining pages in the unprotected cache to the protected cache. Simple experiments have been performed to observe the impact of page mappings on the vulnerability and the runtime. Figure 7 shows the vulnerability reduction when each page is mapped from the unprotected cache to the protected cache in a PPC in descending order of page vulnerability, as shown in Figure 2.

However, reducing vulnerability can be contrary to performance improvement. For example, to reduce the vulnerability of data, it should not remain in the cache for long. It is better to evict and reload the reused data to reduce the vulnerability, but this may degrade performance. Figure 7 shows the

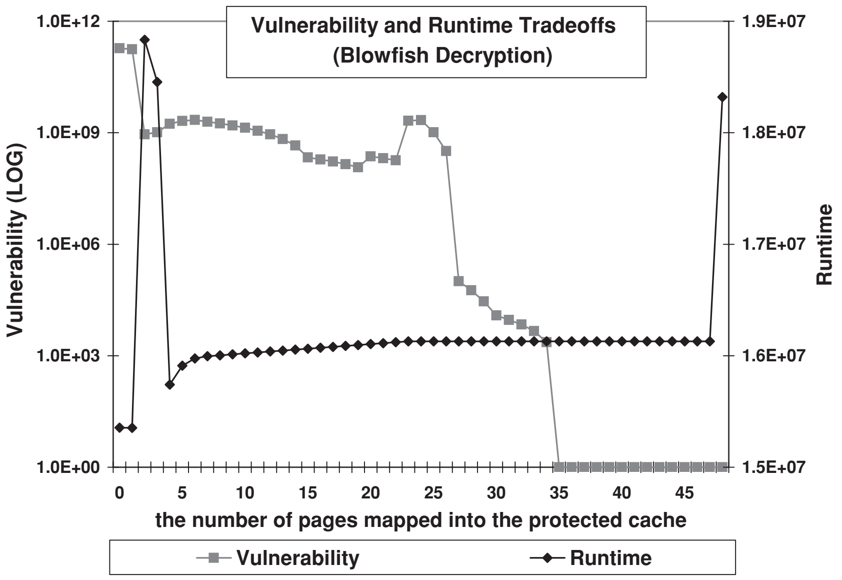


Fig. 7. Vulnerability and runtime with an increase of mapping pages into the protected cache.

trade-off between vulnerability and the runtime. Page partitions can incur significant runtime overheads while reducing vulnerability. Therefore, there is a fundamental trade-off between performance improvement and vulnerability reduction in page partitions for PPC architectures. Thus, our partitioning heuristics will find the interesting partitions while moving pages with higher vulnerability than the others from the unprotected cache to the protected cache, one by one under the performance constraint.

5. PARTITIONING TECHNIQUES

In this section, we present two traditional page partitioning techniques (Section 5.1), and our heuristic partitioning techniques (Section 5.2) for PPC architectures.

5.1 Traditional Page Partitioning Techniques

Our first attempt was to apply generic search algorithms, (1) Monte Carlo method (MC), and (2) genetic algorithm (GA), to explore the solution space. We represent a page mapping by an N bit number, such that if the i^{th} bit of the page mapping is 1, then the i^{th} page is mapped to the protected cache in a PPC. Thus, the page mapping 00..0 represents the default case, when all pages are mapped to the unprotected cache, and the page mapping 11..1 represents the case when all pages are mapped to the protected cache.

5.1.1 *Monte Carlo Method.* Monte Carlo (MC) algorithms are nondeterministic simulation methods, which usually exploit pseudorandom numbers. They are widely used in simulations with a large number degrees of freedom and uncertainty. For each exploration in the MC method, we generate each

```
PPExplore (rPenalty, eWidth, pCount)
$pageMap0 = 0...0$
$< runtime, power, vulnerability >= simulate(pageMap0)$
$config0 = (pageMap0, runtime, power, vulnerability)$
for $(k = 0; k < eWidth; k++)$
  $bestConfigs.insert(config0)$
endFor
for $(;;)$
  $newBestConfigs = bestConfigs$
  for $(i = 0; i < eWidth; i++)$
    for $(j = 0; j < pCount; j++)$
      $testConfig.pageMap = addPage(newBestConfigs[i].pageMap, j)$
      $< runtime, power, vulnerability >= simulate(testConfig.pageMap)$
      if $(runtime < config0.runtime \times \frac{100+rPenalty}{100})$
        if $(vulnerability < newBestConfigs[0].vulnerability)$
          $newBestConfigs.insert(testConfig.pageMap, runtime, power, vulnerability)$
        endIf
      endIf
    endFor
  endFor
  for $(i = newBestConfigs.length(); i > eWidth; i--)$
    $newBestConfigs.delete[i-1]$
  endFor
  if $(newBestConfigs[0].vulnerability < bestConfigs[0].vulnerability)$
    $bestConfigs = newBestConfigs$
  else break;
  endIf
endFor
```

Fig. 8. PPExplore: an exploration algorithm for page partitioning.

bit of the page mapping, 0 or 1, with pseudorandom numbers. Through the simulation, the page mapping is then evaluated with respect to vulnerability, performance, and energy consumption. We then consider the sequence with minimal vulnerability under the runtime constraint.

5.1.2 *Genetic Algorithm.* Genetic algorithms (GA) are adaptive search algorithms using evolutionary ideas such as mutations and crossovers (recombination). Initially, we form a randomly generated sequence, representing a page mapping. At each successive generation, the superior sequences in terms of the vulnerability under the performance constraint are selected as the evolutionary page mappings through the simulation, where vulnerability, performance, and energy consumption are evaluated. In order to generate the next sequence, we implemented two GA operations, for example, mutation and crossover operations. For the mutation operation, a simple pseudorandom number indicates whether each bit in a sequence is modified or not. For the crossover operation, one point is selected in the current sequences and they are swapped on page mappings to the next generated sequences. Then we consider the sequence with least vulnerability under the runtime constraint.

5.2 Customized Page Partitioning Techniques

Our page partitioning techniques employ the vulnerability metric to estimate the failure rate, and they are customized to find a page partition with minimal vulnerability under the runtime constraint.

5.2.1 *PPExplore—Page Partitioning Exploration.* Figure 8 outlines our PPExplore partitioning algorithm, which starts from the case when no page is mapped to the protected cache (all pages are mapped into the unprotected cache) which is the default case. In each step, pages are moved from the unprotected cache to the protected cache, each partition is evaluated, and the best page partition in terms of the vulnerability reduction under the runtime penalty is selected to be mapped into the protected cache in a PPC. Our page partitioning algorithm uses two parameters: (1) allowable runtime penalty (*rPenalty*), and (2) exploration width (*eWidth*), that is, how many partitions are maintained as best configurations for the whole exploration. PPExplore uses *pCount*, the number of pages in a benchmark, and searches for page mappings that will suffer no more than the specified runtime penalty, while trying to minimize the vulnerability. PPExplore maintains a set of best page mappings found so far (line 05) in *bestConfigs*, sorted in order of the vulnerability. After initialization, the algorithm goes into a forever loop in line 07. It takes each existing best solution and tries to improve it by mapping a page to the protected cache, and by evaluating a new page map in terms of runtime, power, and vulnerability (lines 11 and 12). If the new page mapping is better than the worst solution in terms of vulnerability in the *newBestConfigs* with runtime satisfied, then the new page mapping is inserted in the ordered list (*newBestConfigs*) according to the vulnerability (lines 13–17). The loop in lines 09–19 is one step of exploration. After each step, the new set of page mappings is trimmed down to the exploration width (lines 20–22). The termination criterion of the exploration is when an exploration step cannot find any better page mapping. In other words, no page can be mapped to the protected cache to improve vulnerability (lines 23, 25) under the runtime penalty. Otherwise, the global collection of the best page mappings are updated (line 24).

PPExplore is very effective in eventually finding interesting partitions with minimal vulnerability, since it explores all the possible page partitions by moving one page at each step from the beginning.

5.2.2 *qPPExplore—quick PPExplore.* Our PPExplore is effective in finding the interesting partitions but its complexity is $O(mN!)$, where N is the number of pages to be explored and $O(m)$ is the complexity of a simulation to evaluate a page partition. PPExplore is expensive since at each step PPExplore tries all possible partitions by mapping a page from the remaining pages at the unprotected cache into the protected cache, and finds a page partition with minimal vulnerability among them. On the contrary, the complexity of qPPExplore is $O(mN)$ as shown in Figure 9 since it selects the partition with the least vulnerability among partitions explored by moving a page from the unprotected cache to the protected cache in descending order of page vulnerability, which satisfies the runtime constraint. Note that each page is queued in the descending order of page vulnerability and thus the page with the highest vulnerability is mapped into the protected cache at each step. *config*0 keeps the runtime, power, and vulnerability when all pages are mapped into the protected cache in a PPC (lines 01–03). Then, qPPExplore explores a partition by mapping the page with the highest vulnerability, and selects this partition as the best (*bestConfig*) if it satisfies the runtime constraint and it has less vulnerability than the least vulnerability so far (lines 07–13). It repeats page partitioning

```
qPPExplore (rPenalty, pCount)
01: $pageMap0 = 0...0$
02: $< runtime, power, vulnerability >= simulate(pageMap0)$
03: $config0 = (pageMap0, runtime, power, vulnerability)$
04: $bestVulnerabilility = vulnerability$
05: $bestConfig = baseConfig = config0$
06: for $(j = (pCount - 1); j > -1; j - -)$
07:    $baseConfig.pageMap = addPage(baseConfig.pageMap, j)$
08:    $< runtime, power, vulnerability >= simulate(baseConfig.pageMap)$
09:    if $(runtime < config0.runtime \times \frac{100+rPenalty}{100})$
10:       if $(vulnerability < bestVulnerability)$
11:          $bestConfig = baseConfig$
12:       endIf
13:    endIf
14: endFor
```

Fig. 9. qPPExplore: a quick exploration algorithm for page partitioning.

and evaluation until all pages are mapped into the protected cache in a PPC (lines 06, 14).

qPPExplore is efficient in terms of the exploration speed to explore the large set of page partitions and is also effective in finding interesting partitions with minimal vulnerability in benchmarks we have studied, as demonstrated in Section 7.

5.2.3 *EPPExplore—Enhanced PPExplore.* EPPExplore enhances our exploration algorithms by combining PPExplore with qPPExplore. PPExplore begins with exploring partitions from the default case: mapping all pages into the unprotected cache, as shown in lines 01–06 in Figure 8, and tries to improve the vulnerability by finding a page with the minimal vulnerability, which is effective but slow for exploring a large set of possible partitions. However, at the initial step, EPPExplore applies qPPExplore to find the best partition in terms of vulnerability under the runtime constraint. From the partition discovered by qPPExplore, EPPExplore applies the algorithm of PPExplore to further reduce the vulnerability under the runtime constraint. Figure 10 shows that lines 06 to 14 are from qPPExplore and lines 15 to 38 are from PPExplore. EPPExplore can explore the page partitions that may not be explored by PPExplore, since PPExplore stops exploring partitions further if it no longer improves the vulnerability. In fact, it is possible to reduce the vulnerability by mapping multiple pages into the protected cache without significantly degrading performance, while mapping one page out of the remaining pages from the unprotected cache to the protected cache in a PPC does not reduce the vulnerability. Figure 7 shows this possible scenario; increasing the number of pages to be mapped into the protected cache does not keep reducing the vulnerability, as shown in Figure 7. In particular, EPPExplore discovers the interesting page partitions first by qPPExplore, and further reduces vulnerability by extensively exploring partitions with an algorithm in PPExplore. The complexity of EPPExplore is between the complexities of qPPExplore and PPExplore.

6. SETUP

In order to demonstrate the effectiveness of our page partitioning heuristics in exploring and discovering the partition with minimal vulnerability at minimal

```
**EPPExplore (rPenalty, eWidth, pCount)**
$pageMap0 = 0...0$
$< runtime, power, vulnerability >= simulate(pageMap0)$
$config0 = (pageMap0, runtime, power, vulnerability)$
$bestVulnerabilility = vulnerability$
$bestConfig = baseConfig = config0$
**for** $(j = (pCount - 1); j > -1; j - -)$
    $baseConfig.pageMap = addPage(baseConfig.pageMap, j)$
    $< runtime, power, vulnerability >= simulate(baseConfig.pageMap)$
    **if** $(runtime < config0.runtime \times \frac{100+rPenalty}{100})$
        **if** $(vulnerability < bestVulnerability)$
            $bestConfig = baseConfig$
        **endIf**
    **endIf**
**endFor**
**for** $(k = 0; k < eWidth; k + +)$
    $bestConfigs.insert(bestConfig)$
**endFor**
**for** $(;;)$
    $newBestConfigs = bestConfigs$
    **for** $(i = 0; i < eWidth; i + +)$
        **for** $(j = 0; j < pCount; j + +)$
            $testConfig.pageMap = addPage(newBestConfigs[i].pageMap, j)$
            $< runtime, power, vulnerability >= simulate(testConfig.pageMap)$
            **if** $(runtime < config0.runtime \times \frac{100+rPenalty}{100})$
                **if** $(vulnerability < newBestConfigs[0].vulnerability)$
                    $newBestConfigs.insert(testConfig.pageMap, runtime, power, vulnerability)$
                **endIf**
            **endIf**
        **endFor**
    **endFor**
    **for** $(i = newBestConfigs.length(); i > eWidth; i - -)$
        $newBestConfigs.delete[i - 1]$
    **endFor**
    **if** $(newBestConfigs[0].vulnerability < bestConfigs[0].vulnerability)$
        $bestConfigs = newBestConfigs$
    **else break;**
    **endIf**
**endFor**
```

Fig. 10. EPPExplore: a combined exploration algorithm of qPPExplore and PPExplore for page partitioning.

power and runtime[1] penalty, we have built an extensive simulation framework. The application is first compiled to generate an executable. The application is then profiled, and the *Page Vulnerability Estimator* calculates the vulnerability of each page accessed by the application. The pages are then sorted according to their vulnerabilities, and then *Page Partitioning Heuristics* partitions and maps the pages to the two caches in the PPC architecture. Through the simulations, *Page Partitioning Heuristics* finds out the page mapping with minimal vulnerability under the runtime constraint. Finally, the executable and the page mapping are provided to the platform, which runs the application and generates outputs such as runtime, energy consumption, and vulnerability.

The platform is modeled using *sim-outorder* simulator from the SimpleScalar toolchain [Burger and Austin 1997]. The simulation parameters have been set up so as to model an HP iPAQ h4600 [Hewlett Packard]-like processor memory

[1]Here runtime and performance are used interchangeably and represent the number of cycles for execution of an application.

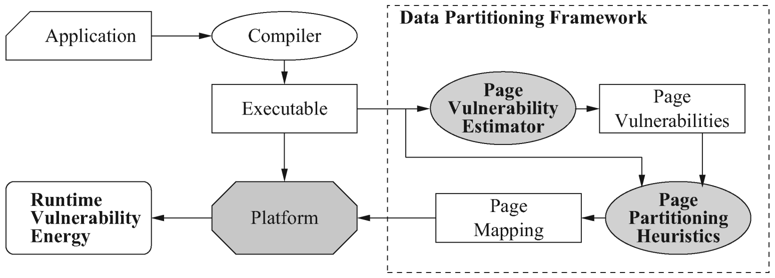

Data Partitioning Framework
Application
Compiler
Executable
Page Vulnerability Estimator
Page Vulnerabilities
Runtime Vulnerability Energy
Platform
Page Mapping
Page Partitioning Heuristics

Fig. 11. Page partitioning exploration framework for PPC architectures.

system. We model a data PPC architecture consisting of a 4 KB unprotected cache and a 256 byte protected cache, and an instruction PPC architecture of a 32 KB unprotected cache and a 2 KB protected cache. Cache parameters are set with a line size of 32 bytes, 4 way set-associativity, and FIFO (first-in first-out) cache replacement policy. This model protects one small cache with an ECC-based technique such as a Hamming code [Pradhan 1996]. The overheads of power and delay for ECC protected caches are estimated and synthesized using CACTI [Shivakumar and Jouppi 2001] and the Synopsys design compiler [Synopsys Inc. 2001].We assume that our ECC-protected caches in PPC architectures are optimized to be accessed within one cycle without overheads in terms of access latency, while ECC-protection incurs energy consumption overheads.

The SimpleScalar *sim-outorder* simulator has been modified to include the vulnerability computation. Thus, the modified *sim-outorder* returns the runtime and vulnerability in cycles for each page partition. To estimate the system energy consumption, we consider the energy consumption of the processor (including the processing pipeline and caches) and the energy consumption of the memory subsystem (including 2 off-chip SDRAMs and external buses). Thus, our model of system energy consumption (E) consists of the energy consumption of the processing core (E_{proc}), that of the caches (E_{cache}), and that of the memory subsystem (E_{mem}). E_{proc} is estimated by multiplying the number of instructions by the power consumption per access, and E_{mem} is estimated by multiplying the number of cache misses by the power overhead per memory access. For the cache energy consumption (E_{cache}), we detail cache access and cache miss into read_access_hit, read_access_miss, write_access_hit, and write_access_miss since each operation results in different ECC events. For example, the energy consumption of read_access_hit is the sum of the access energy consumption and the energy consumption of ECC decoding while the energy consumption of read_access_miss is the sum of the access energy consumption and the energy consumptions of ECC decoding as well as ECC encoding. So the energy consumption model for the cache is $E_{cache} = RH\times d+RM\times(d+e)+WH\times e+WM\times(d+e) = (RH\times d+RM\times d+WH\times d+WM\times d-WH\times d)+(RM\times e+WM\times e+WH\times e) = A\times d+M\times e+WH\times(e-d)$, where RH is the number of read accesses and hits, RM is the number of read accesses but misses, WH is the number of write accesses and hits, WM is the number of write accesses but misses, $A(=RH+RM+WH+WM)$ is the number of cache accesses, M ($= RM + WM$) is the number of cache misses, and d and e are energy consumption of ECC decoding and ECC encoding, respectively.

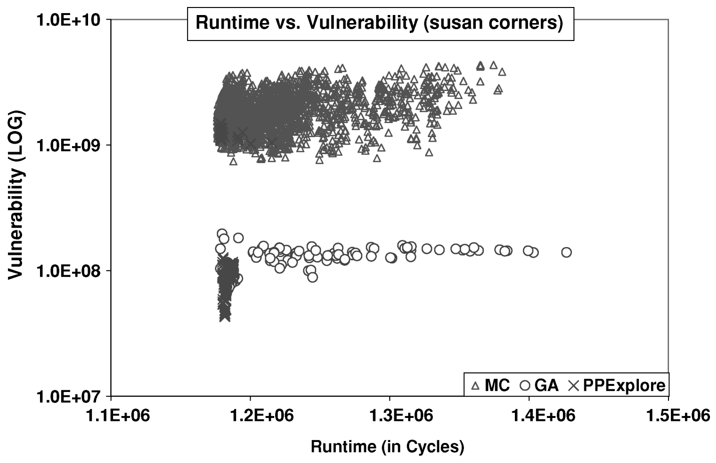
HP iPAQ [Hewlett Packard] is a wireless handheld device, and MiBench is the set of benchmarks that are representatives of applications that run on wireless handheld devices [Guthaus et al. 2001]. The MiBench suite is therefore the right set of benchmarks to run on the iPAQ; and we choose them. However, we pick only those benchmarks in which the runtime difference between the default case when all data is mapped to the 4 KB unprotected cache, and the case when all data is mapped to the 256 byte protected cache in the data PPC is more than 5%. Similarly, we select benchmarks for the instruction PPC. This is to avoid benchmarks for which only the small protected cache is enough. Note that although some of the benchmarks in MiBench are multimedia applications (for which an obvious data partitioning exists), we use our heuristics to partition the data and instructions of all applications in the selected benchmark suite.

7. EXPERIMENTAL RESULTS

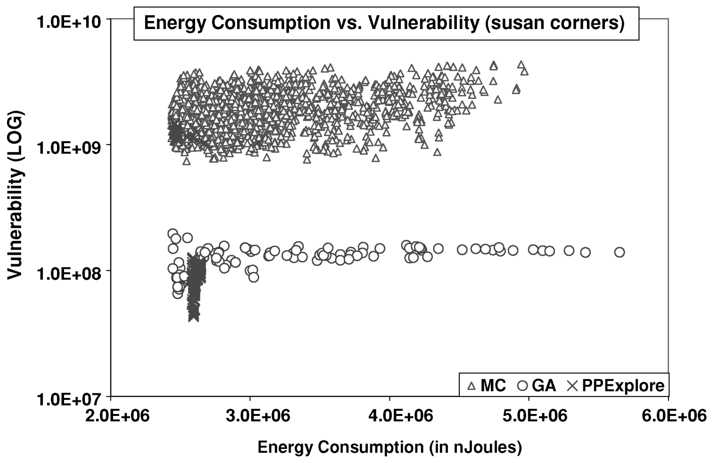
7.1 Comparison of Exploration Algorithms

We bring out the details of exploration using MC (Monte Carlo), GA (genetic algorithm), and PPExplore with data PPCs over the *susan corners* benchmark, when PPExplore is configured for 5% performance penalty, and exploration width, 2. Figures 12(a) and 12(b) plot the runtime, energy consumption, and vulnerability of the page partitions searched by MC, GA, and PPExplore. Note that the y-axis in these graphs—the vulnerability scale—is logarithmic. The most important observation that we make from these graphs is that PPExplore searches much more useful page mappings (low vulnerability), as compared to MC and GA. We allow each technique to explore 1900 page mappings. Thus, in total there are 5700 page mappings. Of them, only 83 are Pareto-optimal. A page mapping is Pareto-optimal, if it is no worse than any other configuration in all three dimensions: runtime, vulnerability, and energy consumption. Out of these 83 Pareto-optimal page mappings, 68 were first drawn from PPExplore searches (82%), 12 came from GA (14%), and only 3 were discovered by MC (4%). This Pareto-optimal observation demonstrates the effectiveness of our algorithm as compared to MC and GA. The main reason for the effectiveness of PPExplore as compared to MC and GA explorations is that PPExplore attempts to explore every possible partition by moving each page in the order of the page vulnerability, and improves the vulnerability from that of the best partitions discovered so far. GA explores more interesting partitions in terms of the vulnerability than MC, since GA attempts to find better partitions from the best one so far, while MC always attempts randomly generated page partitions.

Figure 13(a) plots the vulnerability reduction ratio as the exploration progresses for MC, GA, and PPExplore. *Vulnerability Ratio* indicates the ratio of the vulnerability of the *default case*—all pages mapped into the unprotected cache—to the vulnerability of the page partition discovered by each exploration algorithm. A ratio greater than 1 implies the reduction of the vulnerability metric. The plot shows that while MC is ineffective, GA improves vulnerability by



(a) Runtime and vulnerability



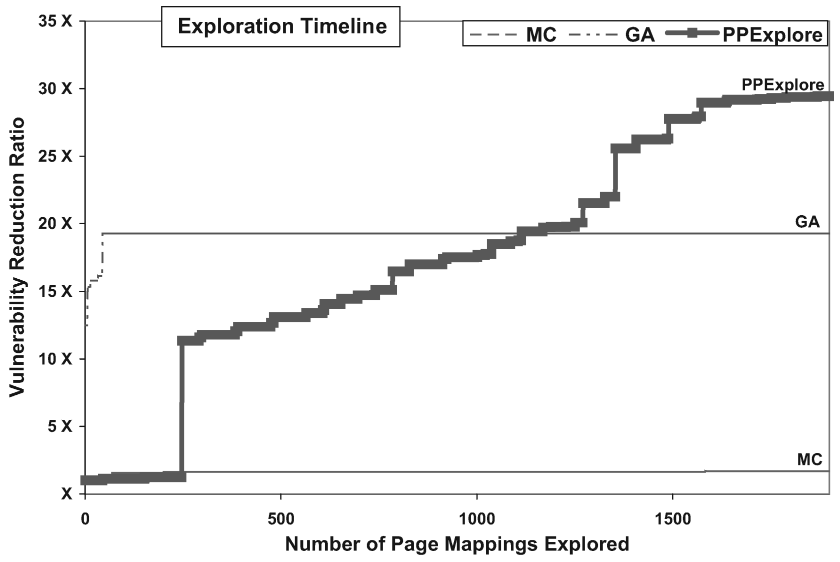
(b) Energy consumption and vulnerability

Fig. 12. Exploration by MC, GA, and PPExplore: PPExplore effectively explores the design space.

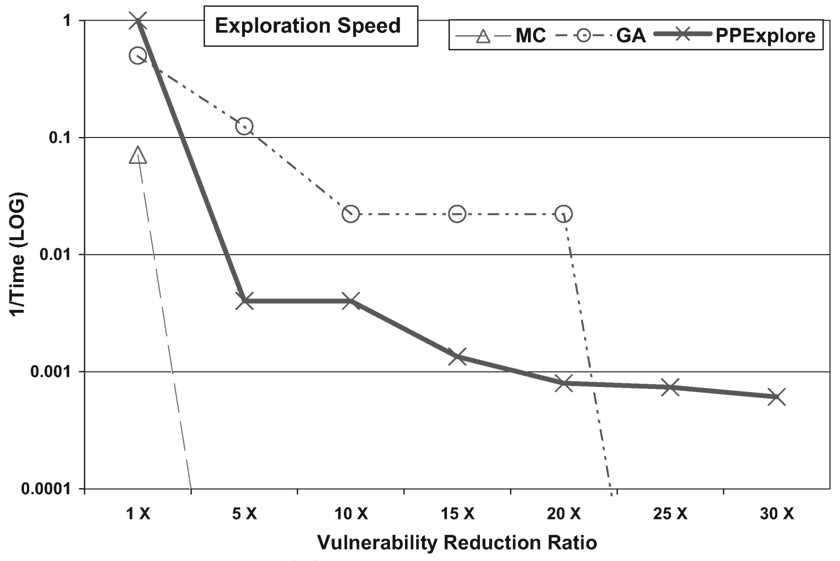
about 20 times, PPExplore continuously finds better page mappings, and is eventually able to reduce vulnerability by about 30 times. Since PPExplore begins with the default page partition, PPExplore improves the vulnerability gradually.

We also compare the speed of the various exploration algorithms. Figure 13(b) plots the speed of exploration: the inverse of the number of page partitions explored to achieve a required vulnerability reduction. The plot shows that MC is quite ineffective. Between GA and PPExplore, GA is a faster approach when low reduction in vulnerability is required, but it is unable to achieve high reduction in vulnerability. This is where our approach is really effective, in terms of the vulnerability reduction.

Further, we run similar experiments over several benchmarks such as a multimedia application (*djpeg*), cryptographic algorithms (*rijndael decryption*



(a) Exploration timeline



(b) Exploration speed

Fig. 13. Exploration by MC, GA, and PPExplore: PPExplore is efficient and eventually finds interesting space.

and *sha*), and a searching algorithm (*stringsearch*), from the MiBench suite. All experimental results demonstrate that our heuristic explorations are more effective than MC and GA in terms of vulnerability reduction under the run-time constraint. For example, PPExplore can find the page partition with about 11 times reduction of the vulnerability for 226 explorations with benchmark, *stringsearch*, while MC and GA find page partitions with less than 6 times

Table I. Evaluation of Vulnerability, Runtime, and Energy Consumption Under No Performance Penalty with Our Partitioning Heuristics for Data PPCs

	Vulnerability Reduction (%)			Runtime Overhead (%)			Energy Overhead (%)		
Benchmarks	PPE	qPPE	EPPE	PPE	qPPE	EPPE	PPE	qPPE	EPPE
susan corners	14.2	6.6	15.1	0.0	0.0	0.0	0.6	0.1	0.1
susan edges	9.8	9.1	20.4	0.0	0.0	0.0	0.5	−0.1	0.0
djpeg	35.2	20.0	31.9	−0.1	−0.5	−0.4	12.9	4.8	7.7
rijndael_dec	0.2	96.7	96.7	0.0	−1.1	−1.1	−0.4	−10.5	−10.5
rijndael_enc	1.6	0.0	1.6	−0.4	0.0	−0.4	−3.8	0.0	−3.8
blowfish_dec	0.6	0.0	0.0	−0.2	0.0	0.0	−1.3	0.0	0.0
blowfish_enc	0.6	0.0	0.6	−0.2	0.0	−0.2	−1.4	0.0	−1.4
fft	14.8	7.7	12.4	0.1	0.0	0.0	2.8	2.4	2.5
sha	21.7	32.7	21.7	0.0	0.0	0.0	−0.1	0.2	−0.1
crc	12.6	97.5	97.5	0.0	−0.5	−0.5	2.3	1.5	1.5
stringsearch	25.5	0.0	34.7	0.0	0.0	0.0	3.5	0.0	−0.1
AVERAGE	12.4	24.6	30.2	−0.1	−0.2	−0.2	1.4	−0.1	−0.4

*PPE = PPExplore, qPPE = qPPExplore, EPPE = EPPExplore.
**Negative value indicates performance improvement or energy reduction.

reduction of the vulnerability for the same number of explorations as PPExplore.

In summary, PPExplore is very effective in exploring page partitions and in finding interesting partitions to reduce the vulnerability with minimal power and performance overheads for PPCs, as compared to the genetic algorithm and Monte Carlo methods.

7.2 Effectiveness of Our Heuristics for a Data PPC

We perform two kinds of experiments to demonstrate the effectiveness of our partitioning heuristics for page mappings into a data PPC for general applications. All the results in Table I and Table II show the vulnerability reduction and runtime/energy overheads of page partitions discovered by our heuristics as compared to those of the default case where all data is mapped into the unprotected 4 KB cache (no protection on the data cache).

In the first set of experiments, we want to find the page partition with the least vulnerability without any performance loss (*rPenalty* = 0%) and exploration width set to 1 (*eWidth* = 1). Table I shows that PPExplore, qPPExplore, and EPPExplore find the page partitions for data PPCs, resulting in vulnerability reductions of 12.4%, 24.6%, and 30.2%, respectively. Since the runtime penalty is set to 0%, the partitions we discovered incur no runtime overhead, as shown in Table I. Table I shows that the data partitions discovered by PPExplore, qPPExplore, and EPPExplore incur the overhead of the system energy consumption by less than 1%. Interestingly, we can find partitions for some benchmarks (e.g., *rijndael decryption* and *crc*), which not only reduce the vulnerability significantly (by about more than 95%) but also improve the performance or the system energy consumption. Note that qPPExplore only explores the number of data pages, and its average over all benchmarks is about 56, while PPExplore explores 627 partitions and EPPExplore explores 401 partitions, on average. Thus, under no runtime penalty, EPPExplore is the best in

Table II. Evaluation of Vulnerability, Runtime, and Energy Consumption Under 5% Performance Penalty with Our Partitioning Heuristics for Data PPCs.

	Vulnerability Reduction (%)			Runtime Overhead (%)			Energy Overhead (%)		
Benchmarks	PPE	qPPE	EPPE	PPE	qPPE	EPPE	PPE	qPPE	EPPE
susan corners	52.3	31.0	43.7	4.4	3.4	3.4	20.0	26.4	26.9
susan edges	67.3	44.1	52.8	4.0	0.0	0.0	41.4	0.1	5.7
djpeg	98.1	20.0	34.5	0.5	−0.5	2.4	1.9	4.8	42.7
rijndael_dec	99.9	99.9	99.9	3.2	−0.4	−0.4	57.0	−3.0	−3.0
rijndael_enc	1.6	0.0	1.6	−0.4	0.0	−0.4	−3.8	0.0	−3.8
blowfish_dec	0.6	0.0	0.0	−0.2	0.0	0.0	−1.3	0.0	0.0
blowfish_enc	0.6	0.0	0.6	−0.2	0.0	−0.2	−1.4	0.0	−1.4
fft	81.5	92.3	94.2	0.0	4.9	4.9	8.1	177.5	178.5
sha	53.7	72.7	77.0	0.1	0.5	0.5	2.1	2.1	2.1
crc	99.9	97.5	97.5	−0.5	−0.5	−0.5	1.5	1.5	1.5
stringsearch	90.8	75.5	77.6	4.9	4.9	4.9	116.5	124.9	130.9
AVERAGE	58.8	48.4	52.7	1.4	1.1	1.3	22.0	30.4	34.5

*PPE = PPExplore, qPPE = qPPExplore, EPPE = EPPExplore.
**Negative value indicates performance improvement or energy reduction.

terms of vulnerability reduction with least overheads of runtime and energy consumption, and qPPExplore searches the least number of partitions explored.

In the next experiment, we allowed 5% performance degradation and an exploration width of 1. Table II presents the vulnerability reduction, the increase in runtime, and the increase in the system energy consumption of the least vulnerable page partitions obtained by PPExplore, qPPExplore, and EPPExplore. Vulnerability reductions achieved by PPExplore, qPPExplore, and EPPExplore are 58.8%, 48.4%, and 52.7%, respectively. Unfortunately, we observe very small vulnerability reductions for benchmarks such as *rijndael encryption*, *blowfish decryption*, and *blowfish encryption*. This result is because some pages are very sensitive to the size of the cache and most pages do not cause any reduction of the vulnerability when they are moved from the unprotected cache to the protected cache in a data PPC. No higher reduction in vulnerability has been observed with any other exploration algorithms such as MC and GA for those benchmarks. Note that qPPExplore only explores the number of data pages (56) while PPExplore explores 1190 partitions, and EPPExplore 460 partitions, on average over all benchmarks. Thus, qPPExplore and EPPExplore are very efficient in terms of exploration speed compared to PPExplore, while PPExplore is best in terms of vulnerability reduction in this set of experiments. Energy consumption overheads are 22.0%, 30.4%, and 34.5% for PPExplore, qPPExplore, and EPPExplore, respectively, while the runtime overheads are less than 5% over all the benchmarks. Thus, even very small runtime degradation allows our heuristics to find page mappings that can significantly reduce vulnerability with minimal overheads. Note that they incur relatively high energy consumption overheads (in particular, the partitions for *fft* and *stringsearch* benchmarks incur more than 100%) since our heuristics do not restrict the energy consumption overhead. We can trade off vulnerability for the reduction of energy consumption. The experimental

Table III. Runtime Overheads and Vulnerability Reductions for Different Data Inputs (Benchmark - *Susan Corners*)

Data Input	Runtime Overhead (%)	Vulnerability Reduction (%)
small (*default*)	4.4	52.3
balloons 1	0.7	35.6
balloons 2	5.4	51.3
columns 1	1.6	25.2
columns 2	3.6	42.6
feep 1	0.2	64.4
large 1	4.4	57.6
large 2	3.7	26.7
AVERAGE	2.8	43.3

results with EPPExplore when both the energy consumption (10%) and the runtime (5%) are limited show that vulnerability reduction is decreased from 58.8% to 48.4% at 2.1% runtime overhead and 7.3% energy consumption overhead, on average.

These experimental results show the effectiveness of PPC architectures and heuristic algorithms compared to conventional ECC-protected caches. Conventional ECC-protected 4KB data caches can reduce 99%[2] of vulnerability while they incur about 15% runtime overhead and 11% energy consumption overhead on average as compared to unprotected 4 KB data caches—the default case. Thus, PPC architectures with our partitioning heuristics can effectively trade off vulnerability for performance improvement and energy reduction.

Note that these experimental results are obtained with sample inputs coming with benchmarks and our profile-based page partitioning heuristics works well if the page mapping of application codes and input data do not change. To observe the impact of different input data, we ran another set of experiments for the benchmark *susan corners* with the best page partition discovered by PPExplore under the 5% runtime penalty for different data inputs. Table III shows the effectiveness of our exploration technique on different data inputs. Other than the default data input (*small*), the vulnerability reduction for seven different data inputs ranges from 25.2% to 64.4% and the average reduction is about 43.3%, which is a little bit less than the vulnerability reduction (52.3%) with the default input in our experiments. These results show that the page partitions for data PPCs discovered by our heuristics can reduce the vulnerability not only with simulated data input but also with different data inputs. Under several different inputs for other benchmarks, we can observe similar results and also observe less reduction of vulnerability in benchmarks with different data inputs.

In summary, the best page partitions discovered by our exploration heuristics (PPExplore) show vulnerability reduction by 48.4% with minimal overheads of runtime (by 2.1%) and energy consumption (by 7.3%) over benchmarks as

[2]The vulnerability of an unprotected cache is estimated in vulnerable cycles during execution, and that of an ECC-protected cache is calculated by multiplying the vulnerability of an unprotected cache with the ratio between the single-bit soft error rate and the double-bit soft error rate, which is about 10^{-2}, since our modeled ECC (a Hamming code) corrects only single-bit soft error.

Table IV. Evaluation of Vulnerability, Runtime, and Energy Consumption Under No Performance Penalty with Our Partitioning Heuristics for Instruction PPCs

	Vulnerability Reduction (%)			Runtime Overhead (%)			Energy Overhead (%)		
Benchmarks	PPE	qPPE	EPPE	PPE	qPPE	EPPE	PPE	qPPE	EPPE
susan corners	17.9	9.6	15.1	0.0	0.0	0.0	2.8	4.4	4.4
susan edges	19.4	8.7	20.3	0.0	0.0	0.0	2.7	0.3	2.6
dijkstra	73.1	12.4	65.2	−0.3	0.0	−0.3	7.0	4.7	4.9
rijndael_dec	11.5	10.8	11.3	0.0	0.0	0.0	1.0	1.0	1.0
rijndael_enc	12.9	12.4	12.9	0.0	0.0	0.0	1.0	1.0	1.0
blowfish_dec	21.0	21.0	21.0	0.0	0.0	0.0	2.0	2.0	2.0
blowfish_enc	20.9	20.9	20.9	0.0	0.0	0.0	2.0	2.0	2.0
fft	9.5	8.6	9.5	0.0	0.0	0.0	0.9	0.7	0.8
stringsearch	15.4	14.2	15.4	0.0	0.0	0.0	0.1	0.1	0.1
AVERAGE	22.4	13.2	21.3	0.0	0.0	0.0	2.2	1.8	2.1

*PPE = PPExplore, qPPE = qPPExplore, EPPE = EPPExplore.
**Negative value indicates performance improvement.

compared to the default case when all data is mapped into the unprotected cache. Our exploration heuristics effectively expand the applicability of data PPCs in general.

7.3 Effectiveness of Our Heuristics for Instruction PPC

To explore the page partitions for instruction PPCs, we employ PPExplore, qPPExplore, and EPPExplore under no performance penalty and 5% performance penalty. For this set of experiments, PPExplore and EPPExplore are configured with exploration width of 2. In these experiments, we added benchmark *dijkstra* instead of *djpeg*, *crc*, and *sha* since they show less than 5% runtime difference between the default case when mapping all instructions into the 32 KB unprotected cache and the case when mapping all instructions into the 2 KB protected cache in an instruction PPC. Under no runtime penalty, Table IV shows that PPExplore discovers the page partitions for instruction PPCs with 22.4% reduction of vulnerability at a cost of 2.2% energy consumption with no runtime overhead as compared to the default case, qPPExplore discovers them with 13.2% reduction of vulnerability and 1.8% energy consumption overhead, and EPPExplore discovers them with 21.3% reduction of vulnerability and 2.1% energy consumption overhead, on average. Under 5% runtime penalty, Table V shows that the page partitions discovered by PPExplore reduce the vulnerability by 49.9% with 1.9% overhead of runtime and 8.2% overhead of energy consumption, the page partitions discovered by qPPExplore reduce the vulnerability by 26.6% with 0.5% runtime overhead and 3.5% energy consumption overhead, the page partitions discovered by EPPExplore reduce the vulnerability by 49.6% with 2.3% runtime overhead and 12.9% energy consumption overhead. In terms of the efficiency (i.e., the number of partitions explored), while qPPExplore evaluates about 56 partitions (the average number of instruction pages among benchmarks), PPExplore evaluates 681 partitions and 1,457 partitions, and EPPExplore explores 593 and 1,299 partitions, under no performance penalty and under 5% performance penalty, respectively. Thus, qPPExplore is the most efficient, and EPPExplore explores fewer partitions

Table V. Evaluation of Vulnerability, Runtime, and Energy Consumption Under 5% Performance Penalty with Our Partitioning Heuristics for Instruction PPCs

	Vulnerability Reduction (%)			Runtime Overhead (%)			Energy Overhead (%)		
Benchmarks	PPE	qPPE	EPPE	PPE	qPPE	EPPE	PPE	qPPE	EPPE
susan corners	48.5	9.6	48.0	4.0	0.0	5.0	16.2	4.4	34.4
susan edges	23.1	8.7	23.3	0.0	0.0	0.1	3.0	0.3	4.9
dijkstra	98.8	96.2	98.0	0.6	2.6	2.7	9.9	14.4	14.6
rijndael_dec	37.4	10.8	36.8	0.4	0.0	0.4	1.7	1.0	2.7
rijndael_enc	34.1	12.4	34.1	0.4	0.0	0.4	1.7	1.0	2.7
blowfish_dec	70.9	29.5	70.9	1.6	0.1	1.6	7.8	2.3	12.7
blowfish_enc	70.8	29.5	70.8	1.6	0.1	1.6	7.8	2.3	12.7
fft	19.1	8.6	18.4	4.7	0.0	4.6	14.1	0.7	14.4
stringsearch	46.5	34.2	46.5	4.3	1.7	4.3	11.3	4.8	16.7
AVERAGE	49.9	26.6	49.6	1.9	0.5	2.3	8.2	3.5	12.9

*PPE = PPExplore, qPPE = qPPExplore, EPPE = EPPExplore.

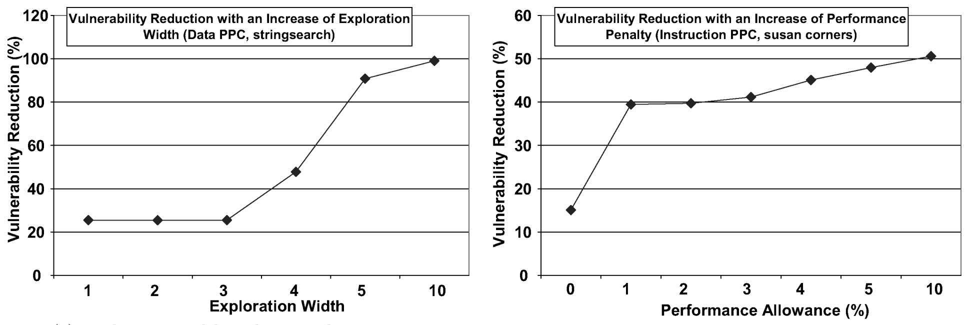
than PPExplore while finding the effective pages in terms of vulnerability reduction close to those of PPExplore. However, the partitions discovered by PPExplore are the most effective in terms of vulnerability reduction with minimal overheads of runtime and energy consumption.

These experimental results show that our partitioning heuristics for instruction PPC architectures can effectively trade off vulnerability for performance improvement and energy reduction. Conventional ECC-protected instruction caches with the size of 32KB incur about 12% performance overhead and 23% energy consumption overhead as compared to the default case while they can reduce 99% vulnerability. The partitions for instruction PPCs discovered by our heuristics can improve vulnerability by about 50% while incuring less overhead of performance and energy consumption than the conventional ECC-protected caches.

In summary, the results that our heuristics obtain are very effective since the instruction PPCs with page partitions discovered by our heuristics can reduce vulnerability by about 50% with less than 2% performance overhead and 8% overhead of the system energy consumption. Our page partitioning heuristics effectively expand the applicability of PPC architectures for instruction caches as well as for data caches.

7.4 Sensitivity of Vulnerability Reduction

We also studied the effectiveness of vulnerability reduction with our heuristics by varying the allowable performance penalty and the exploration width. Figure 14(a) shows that as we increase the exploration width from 1 to 10 with PPExplore, the vulnerability reduction increases with the benchmark *stringsearch*, on data PPCs. Note that this experiment limits the runtime penalty to 0% and is very effective (up to 99% vulnerability reduction under no performance penalty). However, increasing the exploration width can increase the number of partitions explored by up to $eWidth$ times more than the case with exploration width 1.



(a) Exploration width with PPExplore in Data PPC (benchmark - *stringsearch*)

(b) Performance penalty with EPPExplore in Instruction PPC (benchmark - *susancorners*)

Fig. 14. Sensitivity of *exploration width* and *performance penalty* to vulnerability reduction.

Figure 14(b) shows that as we increase the allowable performance penalty, from 1% to 10%, the vulnerability reduction of benchmark *susan corners*, on instruction PPC increases. Note that allowing a greater performance penalty in our heuristics incurs more overhead in terms of performance and energy consumption. However, this performance penalty parameter can increase the effectiveness of finding interesting partitions with least vulnerability, as shown in Figure 14(b).

In summary, when increasing the exploration width and the allowable runtime penalty in our heuristic algorithms such as PPExplore and EPPExplore, increase of vulnerability reduction has been observed in most benchmarks for instruction and data PPCs. Thus, we can definitely trade off exploration time for vulnerability reduction of applications.

8. SUMMARY

Owing to incessant technology scaling, soft errors, especially in caches, are becoming a critical design concern for the reliability of embedded systems. Partially protected cache (PPC) architecture has been proposed as an effective microarchitectural means of improving the system reliability with minimal power and performance penalty for resource-constrained embedded systems. However, the challenge is in partitioning pages among the two caches in a PPC. While page partitioning schemes have been proposed for multimedia applications, there is no page partitioning scheme for data PPCs not for instruction PPCs in general. The page partitioning space is huge, and existing random techniques are unable to identify and explore the page partitions that lead to low vulnerability. In this article, we develop page partitioning heuristics such as PPExplore, qPPExplore, and EPPExplore at design time that effectively and efficiently find page partitions resulting in, on average, 48% reduction in vulnerability (failure rate) at only 2% performance and 7% energy penalty for data PPCs, and 50% reduction in vulnerability at only 2% performance and 8% energy penalty for instruction PPCs over benchmarks.

The main contribution of our partitioning heuristics is that they increase the applicability of PPC architectures and establish PPC as the hardware/software hybrid solution of choice to improve the reliability of cache-based architectures.

Our future work includes intelligent schemes to improve the page partitioning in PPCs and dynamic schemes to relocate page partitions at runtime. Also, we plan to investigate partitioning techniques for more than two caches with different levels of protection for PPC architectures.

REFERENCES

ANGHEL, L. AND NICOLAIDIS, M. 2000. Cost reduction and evaluation of a temporary faults detecting technique. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*. 591–597.

ASADI, G.-H., SRIDHARAN, V., TAHOORI, M. B., AND KAELI, D. 2005. Balancing performance and reliability in the memory hierarchy. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 269–279.

BAUMANN, R. 2005. Soft errors in advanced computer systems. *IEEE Des. Test Comput.* 258–266.

BAZE, M. P., BUCHNER, S. P., AND MCMORROW, D. 2000. A digital CMOS design technique for SEU hardening. *IEEE Trans. Nucl. Sci. 47,* 6, 2603–2608.

BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, version 2.0. *SIGARCH Comput. Arch. News 25,* 3, 13–25.

CHEN, G., KANDEMIR, M., IRWIN, M. J., AND MEMIK, G. 2005. Compiler-directed selective data protection against soft errors. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*.

ERNST, D., KIM, N. S., DAS, S., PANT, S., RAO, R., PHAM, T., ZIESLER, C., BLAAUW, D., AUSTIN, T., FLAUTNER, K., AND MUDGE, T. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 7–13.

GAISLER, J. 1997. Evaluation of a 32-bit microprocessor with built in concurrent error-detection. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Workshop on Workload Characterization*. 3–14.

HAZUCHA, P. AND SVENSSON, C. 2000. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. Nucl. Sci. 47,* 6, 2586–2594.

Hewlett Packard. *HP iPAQ h4000 Series — System Specifications*. Hewlett Packard.

HU, J. S., LI, F., DEGALAHAL, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. J. 2005. Compiler-directed instruction duplication for soft error detection. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*. 1056–1057.

ITRS. 2005. *International Technology Roadmap for Semiconductors 2005 Executive Summary*. http://www.itrs.net/Links/2005ITRS/ExecSum2005.pdf.

KIM, S. 2006. Area-efficient error protection for caches. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*. 1282–1287.

KRISHNAMOHAN, S. AND MAHAPATRA, N. R. 2004. An efficient error-masking technique for improving the soft-error robustness of static CMOS circuits. In *Proceedings of the IEEE International SOC Conference (SOCC)*. 227–230.

KRUEGER, D., FRANCOM, E., AND LANGSDORF, J. 2008. Circuit design for voltage scaling and SER immunity on a quad-core Itanium processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. 94–95.

LEE, K., SHRIVASTAVA, A., ISSENIN, I., DUTT, N., AND VENKATASUBRAMANIAN, N. 2006. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 411–420.

LEE, K., SHRIVASTAVA, A., ISSENIN, I., DUTT, N., AND VENKATASUBRAMANIAN, N. 2009. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Trans. VLSI Syst. 17*, 9, 1343–1347.

LI, J.-F. AND HUANG, Y.-J. 2005. An error detection and correction scheme for RAMs with partial-write function. In *Proceedings of the IEEE International Workshop on Memory Technology, Design and Testing (MTDT)*. 115–120.

Li, L., Degalahal, V., Vijaykrishnan, N., Kandemir, M., and Irwin, M. J. 2004. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. 132–137.

Liden, P., Dahlgren, P., Johansson, R., and Karlsson, J. 1994. On latching probability of particle induced transients in combinational networks. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS)*.

Lucchetti, D., Reinhardt, S. K., and Chen, P. M. 2005. ExtraVirt: Detecting and recovering from transient processor faults. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

Mastipuram, R. and Wee, E. C. 2004. *Soft Errors' Impact on System Reliability*. http://www.edn.com/article/CA454636.

Mitra, S., Seifert, N., Zhang, M., Shi, Q., and Kim, K. S. 2005. Robust system design with built-in soft-error resilience. *IEEE Computer 38,* 2, 43–52.

Mohanram, K. and Touba, N. A. 2003. Partial error masking to reduce soft error failure rate in logic circuits. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*. 433–440.

Mohr, K. and Clark, L. 2006. Delay and area efficient first-level cache soft error detection and correction. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*.

Mukherjee, S. S., Emer, J., Fossum, T., and Reinhardt, S. K. 2004. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. 37–42.

Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 29–40.

Musseau, O. 1996. Single-event effects in SOI technologies and devices. *IEEE Trans. Nucl. Sci. 43,* 2, 603–613.

Nicolaidis, M. 1999. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. 86.

Nieuwland, A. K., Jasarevic, S., and Jerin, G. 2006. Combinational logic soft error analysis and protection. In *Proceedings of the IEEE International Symposium on On-Line Testing (IOLTS)*. 99–104.

Phelan, R. 2003. Addressing soft errors in ARM core-based designs. Tech. rep., ARM.

Pradhan, D. K. 1996. *Fault-Tolerant Computer System Design*. Prentice Hall.

Quach, N. 2000. High availability and reliability in the Itanium processor. *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 61–69.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. 2005a. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I., and Mukherjee, S. S. 2005b. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the International Symposium on Computer Architecture*.

Roche, P., Gasiot, G., Forbes, K., Oapos, Sullivan, V., and Ferlet, V. 2003. Comparisons of soft error rate for SRAMs in commercial SOI and bulk below the 130-nm technology node. *IEEE Trans. Nucl. Sci. 50,* 6.

Shivakumar, P. and Jouppi, N. 2001. CACTI 3.0: An integrated cache timing, power, and area model. Tech. rep. 2001/2, WRL.

Shivakumar, P., Kistler, M., Keckler, S., Burger, D., and Alvisi, L. 2002. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 389–398.

Stackhouse, B., Cherkauer, B., Gowan, M., Gronowski, P., and Lyles, C. 2008. A 65 nm 2-billion-transistor quad-core Itanium processor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. 92–93, 598.

Sugihara, M. 2008. SEU vulnerability of multiprocessor systems and task scheduling for heterogeneous multiprocessor systems. In *Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED)*. 757–762.

SUGIHARA, M., ISHIHARA, T., AND MURAKAMI, K. 2007. Task scheduling for reliable cache architectures of multiprocessor systems. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*. 1490–1495.

SYNOPSYS INC. 2001. *Design Compiler Reference Manual*. Synopsys Inc., Mountain View, CA.

WANG, S., HU, J., AND ZIAVRAS, S. G. 2006. On the characterization of data cache vulnerability in high-performance embedded microprocessors. In *Proceedings of the IEEE International Conference on Embedded Computer Systems, Architecture, Modeling, and Simulation (SAMOS)*. 14–20.

WROBEL, F., PALAU, J. M., CALVET, M. C., BERSILLON, O., AND DUARTE, H. 2001. Simulation of nucleon-induced nuclear reactions in a simplified SRAM structure: Scaling effects on SEU and MBU cross sections. *IEEE Trans. Nucl. Sci. 48,* 6, 1946–1952.

ZHANG, W. 2005a. Computing cache vulnerability to transient errors and its implication. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*. 427–435.

ZHANG, W. 2005b. Replication cache: A small fully associative cache to improve data cache reliability. *IEEE Computers 54,* 12, 1547–1555.

ZHANG, W., GURUMURTHI, S., KANDEMIR, M., AND SIVASUBRAMANIAM, A. 2003. ICR: In-cache replication for enhancing data cache reliability. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

ZORIAN, Y., VARDANIAN, V. A., ALEKSANYAN, K., AND AMIRKHANYAN, K. 2005. Impact of soft error challenge on SoC design. In *Proceedings of the IEEE International Symposium on On-Line Testing (IOLTS)*. 63–68.

Received November 2008; revised April 2009, March 2010; accepted May 2010