Considerations for Mondriaan-like Systems

Emmett Witchel
Department of Computer Science, The University of Texas at Austin
witchel@cs.utexas.edu

Abstract

Mondriaan memory protection is a hardware/software system that provides efficient fine-grained memory protection. Other researchers have embraced the Mondriaan design as an efficient way to associate metadata with each 32-bit word of memory. However, the Mondriaan design is efficient only when the metadata has certain properties. This paper tries to clarify when a Mondriaan-like design is appropriate for a particular problem. It explains how to reason about the space overhead of a Mondriaan design and identifies the significant time overheads as refills to the on-chip metadata cache and the time for software to encode and write metadata table entries.

1 Introduction

Mondriaan memory protection (MMP) is hardware/-software co-design for fine-grained memory protection. Like page tables, the heart of MMP is a set of hardware structures and software-written data structures that efficiently associate protection metadata with user data. Other researchers have used Mondriaan-like structures when they need to efficiently associate non-protection metadata with user data [ZKDK08, CMvPT07]. However, MMP is only efficient under certain assumptions about the metadata and data. This paper tries to clarify these assumptions to guide researchers in when MMP can be useful to address their problems.

While the computer science publication system is effective at creating incentives for researchers to publish innovative results, it is less effective at encouraging researchers to reflect on, and publicly critique, their own work. Students of the field are often left wondering why a promising sounding idea was left unimplemented. Or they wonder why certain ideas from an early paper on a subject are left out of follow-on work. Did those ideas fail or were they simply not explored?

While journals provide some outlet to summarize the progress of a research project, they often default to extended versions of conference papers. This paper is much shorter than a journal paper and tries to convey insights and experience, rather than rigorous quantitative evidence for its conclusions.

While providing a compact summary of MMP research, this paper highlights the assumptions made by various MMP implementations that are required for high performance. These assumptions do not always apply to systems developed by other researchers. The purpose of this paper is to allow other researchers to quickly determine if their application is likely to perform well with MMP-like hardware or what they would have to modify to make it perform well.

This paper discusses the interplay between the following design decisions.

1. **Space overhead.** The space overhead for MMP is approximately the average number of metadata bits per data item. MMP keeps space overhead low by storing 2 bits of protection information per 32-bit word (approximately a 6% overhead). Tables can be encoded for greater space efficiency if there are long stretches of memory with identical metadata values.
2. **PLB reach.** MMP includes an on-chip associative memory for its metadata called the protection lookaside buffer (PLB). For the PLB hardware to be an effective cache, the metadata must have particular properties, either much of it is coarse-grained, or it has long segments with identical metadata values.
3. **Software overheads.** MMP requires system software to write the metadata tables. The meta-

data format must be simple enough and written infrequently enough to prevent software from significantly reducing performance.

2 MMP history

MMP started in 2002 as follow-on work to low-power data caches [WLAA01]. Our idea was to automatically migrate unused program data to a portion of the cache/memory hierarchy that requires lower power to maintain state. To track program objects, which tend to be small and not naturally aligned, we needed a data structure. The data structure would be written by software and read by hardware because we thought the hardware would make frequent decisions about what data belongs in high-power fast memory and what can reside in low-power slow memory.

During the design of the hardware data structure, we realized that solving the basic problem of having hardware track user-defined data structures was more profound than the application of moving data to save energy. We soon left that motivation and chose fine-grained protection. The plugin model for program functionality extension made the motivation for fine-grained protection clear. Programs (like the OS and a web browser) load user-supplied code directly into their address space to extend functionality. The problem with this approach is that a bug can crash the entire program—plugins are fast, but not safe. Fine-grained protection can restore the safety without reducing the speed of the plugin extensibility model.

The first MMP paper focused on the format of the hardware tables [WCA02]. This paper is most often cited by those interested in MMP. It introduces the basic idea of MMP and presents both a simple table format and a more advanced table format, a technical innovation explained in §3.2. It also contains a design for fine-grained memory remapping, which allows a user to stitch together bits of memory into a contiguous buffer. The design for remapping is a bit complicated, but provides good support for zero-copy networking. The issues for supporting protection dominated the project after this paper and the remapping was dropped, simply for lack of space.

The follow-on paper [WA03] describes how the OS support for fine-grained protection domains would work and how to support safe calling between protection domains. Though our experience was limited at the time, much of our design ended up in our final implementation. My thesis [Wit04] continued to refine the OS support and added ideas for protecting the stack. MMP culminated in an SOSP paper [WRA05], which is the most complete implementation of the system, though it is not often cited. Most of the interest in MMP comes from computer architects, many of whom do not regularly read the proceedings of SOSP.

3 MMP technical summary

This section provides a high-level summary of how MMP works, with a focus on how MMP-like hardware would be used for other applications. The three main features of MMP are memory protection, protected cross-domain calling, and stack protection. The feature most attractive for other uses is a generalization of memory protection, which associates metadata with every word of user data. This section focuses on the general design of that protection mechanism.

3.1 CPU modifications

MMP consists of hardware and software to provide fine-grained memory protection. MMP modifies the processor pipeline to check permissions on every load, store, and instruction fetch. MMP is designed to be simple enough to allow an efficient implementation for modern processors, but powerful enough to allow a variety of software services to be built on top of it. The permissions information managed by MMP could be generalized to any metadata.

Figure 1 shows the basics of the MMP hardware. MMP adds a *protection lookaside buffer* (PLB) that is an associative memory, like a TLB. The PLB caches entries of a memory-resident permissions (or metadata) table, just as a TLB caches entries of a memory-resident page table. The PLB is indexed by virtual address. MMP also adds two registers, the protection domain ID, and a pointer to the base of the permissions table. The protection domain ID identifies the protection (or metadata) context of a particular kernel thread to the PLB, just as an address space identifier identifies a kernel thread to a TLB.

The protection domain ID register is not necessary, but without it, the entire PLB must be flushed on

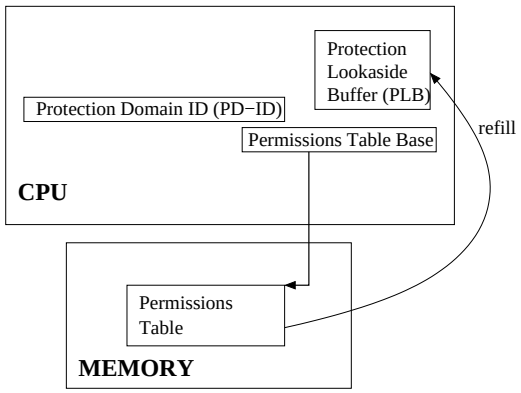

Figure 1: The major components of the Mondriaan memory protection system.

every domain switch. For some kinds of metadata, this might be acceptable. For example, if each kernel thread is its own domain, then domain switches only happen on context switches, which are relatively rare. In this case, the protection domain ID can be dispensed with, just as the x86 does not tag its TLB. However, many consider the lack of tags in the x86 TLB a major design flaw.

On a memory reference, the processor checks the PLB for permissions (or performs whatever metadata check is specified by a system using an MMP-like structure). If the PLB does not have the permissions information, either hardware or software looks it up in the permissions table residing in memory. The reload mechanism caches the matching entry from the permissions table in the PLB, and possibly writes it to the address register sidecar registers. Sidecar registers were a feature of the early MMP design. They are a cache for the PLB, meant to reduce the energy cost of indexing the PLB. They are simply an energy optimization, and because they are inessential, we do not mention them further.

Just like the TLB, the PLB is indexed by virtual address. The PLB lookup can happen in parallel with address translation because MMP stores its metadata per virtual address. Virtual addresses that alias to the same physical address can have different permissions values in MMP.

One of the hardware efficiencies of MMP is that the permissions check can start early in the pipeline and can overlap most of the address translation stages and computational steps of the pipeline. The permissions check need finish only before instruction retirement.

3.2 Permissions table

The MMP protection table represents each *user segment*, using one or more *table segments*. A user segment is a contiguous run of memory words with a single permissions value that has some meaning to the user. For example, a memory block returned from `kmalloc` would be a user segment. User segments start at any word boundary and do not have to be aligned. A table segment is a unit of permissions representation convenient for the permissions table. MMP is not efficient for arbitrary user segments, it assumes certain properties of user segments to achieve efficient execution (§3.2).

System software converts user segments into table entries when permissions are set on a memory region. As explained in §4.3, the frequency and complexity of transforming user segments into table segments determines whether software is an appropriate choice for encoding table segments. Some table entry formats are inefficient for software to write at the update rates required of applications.

Effective address (bits 31−0)

Root Index (10)	Mid Index (10)	Leaf Index (6)	Leaf Offset (6)
Bits (31−22)	Bits (21−12)	Bits (11−6)	Bits (5−0)

Figure 2: How an address indexes the trie.

MMP uses a trie to store metadata, just like a page table. The top bits of an address index into a table, whose entry can be a pointer to another table which is indexed by the most significant bits remaining in the address.

Figure 2 shows which bits of a 32-bit virtual address are used to index a particular level of the MMP permissions table trie. Three loads are sufficient to find the metadata for any user 32-bit word. The lookup algorithm (that can be implemented in software or hardware, just like a TLB) is shown in pseudo-code in Figure 3. The root table has 1024 entries, each of which maps a 4 MB block. Entries in the mid-level table map 4 KB blocks. The leaf level tables have 64 entries, each providing individual permissions for at least 16 four-byte words. The table indices are expanded for 64-bit address spaces [Wit04].

```
PERM_ENTRY lookup(addr_t addr) {
  PERM_ENTRY e = root[addr >> 22];
  if(is_tbl_ptr(e)) {
    PERM_TABLE* mid = e<<2;
    e = mid[(addr >> 12) & 0x3FF];
    if(is_tbl_ptr(e)) {
      PERM_TABLE* leaf = e<<2;
      e = leaf[(addr >> 6) & 0x3F];
    }
  }
  return e;
}
```

Figure 3: Pseudo-code for the trie table lookup algorithm. The table is indexed with an address and returns a permissions table entry. The base of the root table is held in a dedicated CPU register. The implementation of `is_tbl_ptr` depends on the encoding of the permission entries.

The granularity of the metadata is determined by the level at which it appears. Each two-bit entry in a leaf table encodes permissions for a user word of memory. At one level higher, each mid-level entry represents permissions for an entire 4KB page. Regions of at least 4KB that share a single permissions value can therefore be represented with less space overhead. This space savings happens regardless of the entry format.

MMP designs have used different permissions entry formats, notably bitmaps and run-length encoded (RLE) entries (shown in Figure 4). Each leaf entry in bitmap format has 16 two-bit values indicating the permissions for each of 16 words. Run-length encoded entries encode permissions as 4 regions with distinct permissions values, dedicating 8 out of 32 bits to metadata.

RLE entries cannot represent arbitrary word-level metadata. They assume that contiguous words have the same metadata value. For MMP's RLE entries, there can be no more than 4 distinct metadata regions in the entry's 16 data words. If each word has a metadata value distinct from its immediate neighbors, then there are 16 metadata regions and that cannot be represented with an RLE entry. Bitmap entries are used as backup in this case.

The permissions data in MMP run-length encoded entries overlaps with previous and succeeding entries. In addition to permissions information for the 16 words, they can also contain permissions for up to 31 words previous and 32 words subsequent to the 16. In the best case a single RLE entry can contain permissions from 5 distinct bitmap entries.

MMP uses RLE entries to overlap permissions information, but they can be used to save space in leaf-level tables. A 32-bit RLE entry can represent permissions information about 79 words. Taking into account alignment, each entry could encode permissions for 64 words instead of 16, bringing down the average space overheads for leaf-level tables from 6% to 1.6%. Doing so would change the lookup algorithm in Figure 3, because the leaf index would require only 4 bits, leaving 8 bits for the leaf offset. This new RLE format would be more restrictive, only allowing 4 permissions regions in every aligned 64 word block.

4 Requirements for good MMP performance

This section distills our observations on the factors salient for a particular instantiation of an MMP-like system to have good performance.

4.1 Space overhead

While physical memory capacity continues to grow at an impressive rate, MMP-like systems consume memory in proportion to the virtual memory used by a process. As processes use more memory, MMP uses more memory to hold the metadata associated with the data. Keeping the size of the metadata tables reasonable is a first order concern for the practicality of the system.

For the simplest Mondriaan system [WCA02], the space overhead of the most fine-grained tables is approximately 6%, for 2 bits of metadata per 32-bit data word. Both bitmaps and run-length encoded entries dedicate two bits of table entry per user word in leaf-level tables that manage permissions for 32-bit words. The run-length encoding could be adjusted for lower space overhead (§3.2). The mid-level entries that manage permissions for 4KB pages specify 2 bits of metadata per aligned 512 bytes of data, for a space overhead of 0.8%.

Mondrix [WRA05] (the application of Mondriaan

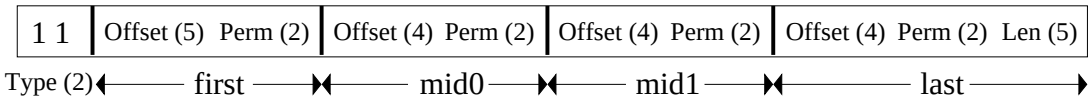

Figure 4: The bit allocation for a run-length encoded (RLE) permission table entry.

memory protection to the Linux kernel), modifies the kernel memory allocator to create larger, aligned data regions. The slab [Bon94] allocator takes a memory page and breaks it into equal sized chunks that are doled out by `kmalloc`, the general-purpose kernel memory allocator. By turning read/write permissions on and off for the entire page, rather than for each individual call to `kmalloc`, Mondrix greatly reduced the space overheads of the permissions. The cost is less memory protection. A read or write into the unallocated area of a page being used as a slab is an error can be detected if the page's permissions are managed on a per-word basis. However, Mondrix forgoes this protection, enabling read and write permissions to the entire page once any of it is used.

Colorama [CMvPT07] uses a Mondriaan design and extends the permission table entries with 12-bit color identifiers that allow the processor to infer synchronization for user data. The color identifiers bring the overhead from 2 bits per 32-bit word to 14 bits per 32-bit word, which is a 44% space overhead. Run-length encoding can bring down this overhead. Using MMP's RLE entry expands the 8 permissions bits to 48 for a 14% space overhead (though keeping entries aligned, which is necessary for a realistic design, would increase the space overhead to nearly 19%). Furthermore, not every data item needs to be colored, only those accessed by multiple threads. The Colorama implementation has a measured overhead of 0–28% space overhead.

Loki [ZKDK08] uses tagged memory to reduce the amount of trusted code in the HiStar operating system. Loki differs from MMP in that the tags are for physical memory. Additionally, Loki maintains two distinct maps, one from physical memory address to tag and another from tag to access permissions.

Loki segregates pages on the basis of whether they need fine-grained tags. Pages with fine-grained tags have 100% space overhead (a 32-bit tag for a 32-bit word), while pages without fine-grained tags (one tag for the entire page) have 0.1% space overhead. The authors see a variable fraction of memory pages that use fine-grained tags, from 3–65%. For this scenario, the fraction of memory pages using fine-grained tags dictates the memory overhead, so the application that uses fine-grained tags for 65% of its pages, experiences a space overhead of 65%. Loki does not use an MMP design for its tags, but the designers note that MMP's RLE entries could save space.

4.2 PLB reach

MMP uses a protection lookaside buffer (PLB) to cache permissions information for data accessed by the CPU, avoiding long walks through the memory resident permissions table. A high hit rate for the PLB is essential for low latency performance. Without a high hit rate, the processor is constantly fetching data from the permissions tables, which increases memory pressure, cache area pressure and decreases the rate at which instructions can retire.

MMP contains several features to enhance the hit rate in the PLB that can be adopted as-is by other projects. The PLB allows different entries to apply to different power-of-two sized ranges. This mechanism allows large granularity entries to co-exist with word-granularity entries (much like super-pages in TLBs). The PLB tags also include protection domain IDs to avoid flushing the PLB on domain switches. Tags are important for Mondrix because its fine-grained protection domains can be crossed as frequently as every 664 cycles [WRA05]. Other applications of MMP might not have such frequent domain crossings.

The main technique for MMP to increase PLB reach is to use large granularity entries (which is done by Mondrix) or run-length encoded entries (e.g., vpr and twolf from SPEC2000 [WCA02]). The PLB miss rate for Mondrix was lower than 1% for all workloads and the execution penalty for PLB refill was less than 4% of execution time because kernel text and data sections are represented with a single entry, and as mentioned in the previous section, the kernel memory allocator was modified to manage protections at the granularity of a page.

When each user allocation for vpr and twolf is protected by inaccessible words at the start and end of the allocation, the system spends 10–20% of its memory references refilling the PLB [WCA02] when using bitmap entries. Run-length encoded entries increase PLB reach by effectively encoding large user segments that share a permissions value with the start of the entry and/or its end. Using run-length encoded entries, memory accesses to the permissions table drop to 7.5% for both SPEC2000 benchmarks. As §3.2 discusses, RLE entries encode overlapping permissions information. A single RLE entry can contain the permissions information from multiple bitmapped entries, eliminating PLB refills.

Because Colorama only monitors shared data accesses, that decreases traffic to the on-chip metadata cache (the Colorama PLB). The Colorama implementation uses run-length encoded entries (called mini-SST entries in the original design [WCA02]), which should be effective at making PLB reach large enough for high performance. Additionally, the authors mention that color metadata could be aggregated into larger granularity chunks, by doing pooled memory allocation.

Loki's support for page-granularity metadata tags is crucial to keeping its runtime overheads low. For one `fork`/`exec` benchmark, page-granularity tags reduces the time overhead from 55% to 1%. It has an 8-entry cache to map from physical page to tag or pointer to a page of fine-grained tags. The fine-grained tags are stored in the CPU's cache. The physical address to tag map does not need to be flushed on a context switch. Loki also has a 32-entry 2-way associative cache that maps tags to permissions. This cache does need to be flushed on context switches, but does not need to be flushed when memory changes tags.

4.3 Software overheads

In an MMP-like design, metadata is managed like page tables are managed, software writes table entries that are read by hardware. Mondrix writes protection tables frequently to protect memory allocations, to protect network packets, and to protect arguments to cross-domain calls. The time for software to write the tables can become a significant performance cost, up to 10% of the kernel execution time in one Mondrix workload.

It is possible that a given application for an MMP-like system will have infrequent metadata updates. Having software encode table entries is a good choice for systems that update metadata infrequently because software is so flexible. However, we found that the only reliable technique for evaluating the cost of the software encoding is to implement it and run it on realistic inputs. The software entry encoding does not need to play a functional role in the system, but it is necessary for benchmarking.

The MMP ASPLOS paper [WCA02] does not evaluate the cost of writing table entries in software as it is a typical hardware evaluation paper that lacks system software support. In its defense, the system software required years of development effort, though effort to develop the table-entry encoder was a small fraction of that time. One unexpected consequence of writing the software to encode table entries is measuring the high runtime cost of writing run-length encoded entries. On one trace of memory protection calls extracted from Mondrix execution, writing run-length encoded entries is three times slower than writing bitmap entries. The run-length encoded entries are slow for software to write because they are complicated to encode, and because they overlap updates to an entry requires complicated logic for breaking and coalescing adjacent entries. While we developed and debugged the code to write run-length encoded entries (a task that required a solid month), we never deployed it in Mondrix because of its poor performance. Also, Mondrix had enough coarse-grained allocations that it did not need run-length encoded entries. Because of our experience with the software, we believe that any MMP implementation with run-length encoded entries will require hardware to encode the table entries.

Run-length encoded entries might be effective for Colorama, because the metadata update rate should be lower than Mondrix's. Mondrix writes the permissions table on memory allocations, and also during data structure processing (e.g., packet reception) and for cross-domain calls. The Colorama implementation measures low allocation rates for some applications (every 129K-288M instructions), and high rates for others, every 2–4K instructions. The authors conservatively assume that every allocation is for colored data, while the true rate for changing the color table

might be lower. The encoding costs for the run-length encoded entries might be an issue if the color tables are actually updated every 2–4K instructions.

Loki's simple data layout can be efficiently written by software. Page-granularity tags are held in an array, and fine-grained tags occupy the same offset in a page as their associated data.

Summary. The trade-offs among space overhead, PLB reach and software overheads are complex for a high-performance MMP-like system. Applying MMP to SPEC2000 and to the Linux kernel resulted in different trade-offs. For projects that only tangentially involve an MMP-like structure, the details of these trade-offs is out of scope. However, a high-level argument for the plausibility of a specific application is necessary to make an argument for the efficiencies of an MMP implementation.

5 Conclusion

The hardware and software designs for Mondriaan memory protection can be used to associate arbitrary metadata with individual user words at reasonable storage and execution time costs. However, keeping those costs limited requires careful design. The original MMP design makes assumptions that follow-on work may violate.

We encourage others to use MMP-like structures, and to include a discussion about space overhead, PLB reach, and software overheads. We hope this paper can act as a guide. The original MMP design limits its space overhead to 6% by using 2 metadata bits for each data word. It increases PLB reach either by using run-length encoded entries or by relying on large user segments. MMP limits software overheads by writing bitmaps in software and run-length encoded entries in hardware.

Acknowledgments

We thank Nickolai Zeldovich and Luis Ceze for their considered comments on a draft of this paper. Thanks also to Owen Hofmann and the anonymous referees for their constructive comments.

References

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[CMvPT07] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *IEEE International Symposium on High Performance Computer Architecture*, 2007.

[WA03] Emmett Witchel and Krste Asanović. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *HotOS*, 2003.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[Wit04] Emmett Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachussetts Institute of Technology, January 2004.

[WLAA01] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[WRA05] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[ZKDK08] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Operating Systems Design and Implementation*, 2008.