A Reuse-Aware Prefetching Scheme for Scratchpad Memory

Jason Cong, Hui Huang, Chunyue Liu and Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, huihuang, liucy, zouyi}@cs.ucla.edu

ABSTRACT

Scratchpad memory (SPM) has been utilized as prefetch buffer in embedded systems and parallel architectures to hide memory access latency. However, the impact of reuse pattern on SPM prefetching has not been fully investigated. In this paper we quantify the impact of reuse on SPM prefetching efficiency and propose a reuse-aware SPM prefetching (*RASP*) scheme. The average performance and energy improvements are 15.9% and 22.0% over cache prefetching, 12.9% and 31.2% over prefetch-only SPM management, 18.5% and 10% over DRDU [1] with SPM prefetching support.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Design Styles—*Cache memories*

General Terms

Algorithm, Performance, Design

Keywords

scratchpad memory, prefetch, reuse

1. INTRODUCTION

Scratchpad memory (SPM) has been widely used in embedded systems and commercial heterogeneous high performance processors like IBM's Cell processor and NVIDIA's GPUs as fast-access storage sitting close to computing logics. Programmers can tune the software manually or through special compiler support to manage SPM explicitly and make better use of the hardware.

It has been shown that SPM can collaborate with D-cache or I-cache to effectively enhance performance and reduce power consumption compared to a pure cache system. This trend has already been reflected on real designs, e.g., NVIDIA's latest Fermi GPU has SPM called "shared memory" which can be partitioned into cache and SPM at configuration points 1:3 or 3:1, with SPM and L1 cache sitting on top of L2 cache. Similarly, the local store in IBM's Cell broadband can be managed as a combination of direct buffers to store access with regular patterns and software-controlled cache as a fall-back solution [2].

A number of works addressing the compiler support for efficient SPM management have been developed. The allocation scheme in SPM can be divided into two categories. The first category is static allocation where data layout in SPM is determined at compile time and will remain fixed throughout program execution. Examples of static SPM allocation schemes include [3], [4], [5] and [6]. Compared with a static scheme, dynamic allocation allows SPM data transfers during execution and hence can better accommodate runtime program requirements. For example, the work in [7] applies loop and data transformation to efficiently reduce the number of data transfers between SPM and main memory. In [8] a compiler-driven approach is presented which partitions the program into code regions and the bring-in/swap-out sets for each region are determined heuristically. In [9] the authors propose a dynamic compiler-directed approach to manage SPM through array live-range partitioning and graph coloring. The SPM buffer allocation approach in [1] is based on memory access pattern analysis to improve data reuse.

In the aforementioned work, SPM has been used to store frequently accessed data for the purpose of performance improvement and energy reduction. In fact, SPM can also be utilized as a prefetch buffer with explicit control over data replacement policy. Compared with conventional cache prefetching, SPM-based prefetching can avoid the scenario in which the data evicted from cache by the newly prefetched data is still "alive," i.e., will be accessed frequently in the near future. An extreme case is that N prefetched elements are mapped to the same set in a direct-mapping cache. Therefore, only the last element will be kept in the cache after prefetching, while the previous $N-1$ data transfers are useless with additional energy overhead. On the other hand, SPM-based prefetching can make a "smart" eviction decision, thereby avoiding such cache prefetch inefficiency.

However, less attention has been given to efficient SPM prefetching in the previous work. For example, the management scheme in [1] only includes initial prefetching operations with no further analysis of prefetching for dynamic data transfers, and program execution may stall due to late SPM buffer update. In fact, prefetching too late to hide the memory access latency will harm the overall performance, while prefetching too early will put stress on the required SPM size to accommodate those data before their first access. The work in [9] prefetches the entire array into SPM before its access with the assumption that the entire array can fit into SPM. While this is usually not the case, e.g., in scientific applications with large input array. In [2], the direct buffers in Cell's local store (SPM) are utilized to support data prefetching with runtime library support. In [10], array prefetching in SPM is managed

DAC 2011, June 5-10, 2011, San Diego, California, USA.

through Markov-chain-based prediction. In [11], SPM is used as a prefetch buffer for video applications by gradually overwriting old data with new data. One common limitation of those works is that SPM prefetching decisions are made independently without considering possible data reuse pattern. For example, [2] and [10] only focus on applications without regular memory reuse, and the scheme proposed in [11] works only for streaming applications.

There exist some unified prefetching and reuse schemes for cache. For example, prefetching instructions in [12] are issued only for the memory references with high probability to be a miss. However, since the work targets a normal cache, the compiler does not have explicit control over data eviction, and cache pollution may still occur. Besides, since the cache block to store the prefetched data is determined by hardware, data layout and eviction set selection are not considered in this work. The same problem also exists in other cache prefetching work such as [13] and [14]; hence, those works cannot be directly applied to SPM prefetching.

To the best of our knowledge, this is the first work to systematically investigate efficient SPM prefetching with the assist of reuse patterns. The contributions of this work are summarized as follows:

(1) We quantify the impact of regular reuse patterns on SPM data prefetching decisions. If the SPM prefetching decisions are made independently of reuse patterns, redundant data transfers to SPM may result in higher energy consumption, as well as larger SPM space demand. We evaluate the impact in our simulation platform and show significant differences in performance and energy between SPM prefetching with/without considering data reuse.

(2) We propose a reuse-aware SPM prefetching scheme, called *RASP*, to hide memory access latency and minimize the number of data transfers from lower-level memory. The concept of *reuse candidate graph* is introduced to guide prefetching decisions. The proposed scheme is evaluated with cache prefetching, prefetch-only SPM management and a DRDU-generated SPM management scheme [1]. The average performance/energy gains over those three schemes are 14.9%/31.2%, 13%/31% and 18.5%/10%, respectively.

2. IMPACT OF REUSE PATTERN ON SPM PREFETCH EFFICIENCY

2.1 SPM Prefetching Without Reuse

With explicit control on data movement, we need to identify the prefetched and evicted data sets, which is essential for an SPM prefetching scheme. The basic implementation is to prefetch data P iterations earlier than its actual access to hide the load latency, where P is the estimated prefetch latency from lower-level memory in terms of loop iteration [15]. In other words, in order to hide the memory access latency, the prefetching instruction of the memory reference set at iteration $i+P+1$ will be issued at iteration i and will replace iteration i's data access set. In this scheme SPM is mainly used as prefetch buffer with size $P+1$. Figure 1 shows a simplified loop kernel code of 429.*mcf* from the *SPEC*2006 suite. At iteration i the newly prefetched data for iteration $i+P+1$, replace $cost[i]$, $head[i]$ and $tail\_potential[i]$ which will not be re-accessed later.

2.2 SPM Prefetching With Regular Reuse Patterns

For programs with regular data reuse patterns, the naive prefetching scheme that simply replaces the data accessed at the current iteration with the data set to be accessed after P iterations is not efficient. More specifically, the data to be prefetched or brought in may already reside in SPM. In this case, duplicate prefetching for the same data from conventional memory will increase the number of issued prefetching instructions as well as the total energy consumption. On the other hand, if the data to be re-accessed in the near future is moved out of SPM, those data need to be re-prefetched into SPM before the next accesses, which also will introduce additional overhead. Figure 2(a) shows the kernel code of 401.*bzip*2 from the *SPEC*2006 benchmark. We can see that iteration $i=8$ is the dividing point where reuse occurs and the prefetching set shrinks by half since the data to be prefetched have already been brought into SPM at an earlier iteration. For example, $fmap[4]$ is brought into SPM as $fmap[i]$ at iteration $i=4$, and is re-accessed as $fmap[i-4]$ at iteration $i=8$. The iteration space after iteration 8 can be seen as a "stable" region, and the prefetching set for any iteration in that region only contains $fmap[i]$.

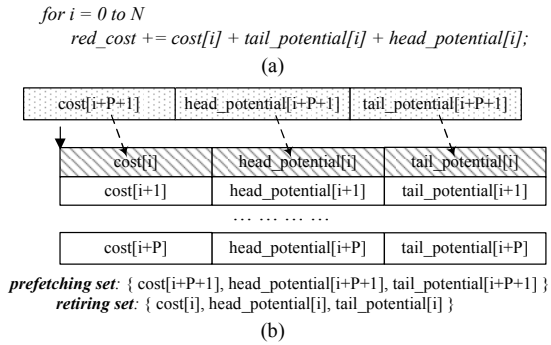

Figure 1: (a) Simplified kernel of 429.mcf. (b) SPM management of 429.mcf.

Figure 2 further illustrates the difference between prefetching schemes with/without considering reuse patterns. The prefetch-only and reuse-aware SPM prefetching schemes are shown in Figure 2(b)(c). In the reuse-aware scheme, the prefetch set at iteration i only contains one element $fmap[i+P+1]$, as $fmap[i+P-3]$ has already been prefetched at iteration $i-4$. The corresponding retiring set only contains $fmap[i-4]$. Compared with the prefetch-only scheme, the number of prefetch instructions issued at each iteration is reduced by 2X, and the associated access to lower-level memory will also be reduced accordingly. In Section 4 we show that compared to the prefetch-only scheme, the reuse-aware prefetch strategy can achieve up to a 42.6% reduction on energy consumption and a 39.3% reduction on execution time.

3. REUSE-AWARE SPM MANAGEMENT

3.1 Preliminaries

DEFINITION 1. [16] *Given a normalized n-level loop nest, suppose there is data dependence between memory reference R_1 at iteration $\vec{i}$ and reference R_2 at iteration $\vec{j}$, then the reuse distance vector $\vec{d}$ is defined as a vector of length n such that $\vec{d}(R_1,R_2) = \vec{j}-\vec{i}$.*

DEFINITION 2. *A reuse candidate graph is a directed graph $G(V_G,E_G)$ where V_G are array references in a uniformly generated set (UGS)*[1] *and each reuse edge $V_s \rightarrow V_d$*

[1] A *uniformly generated set* is a set of affine references of the same array with the same access matrix.

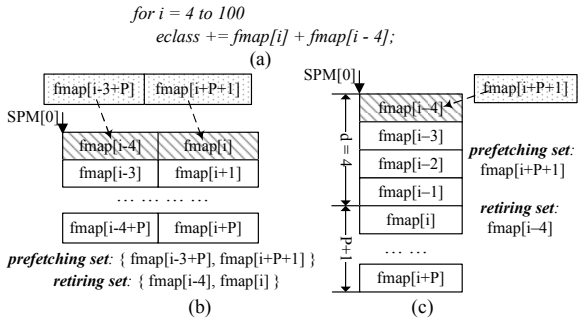

Figure 2: (a) Simplified kernel of 401.bzip2. (b) Prefetch-only SPM management of 401.bzip2. (c) Reuse-aware SPM prefetching scheme of 401.bzip2.

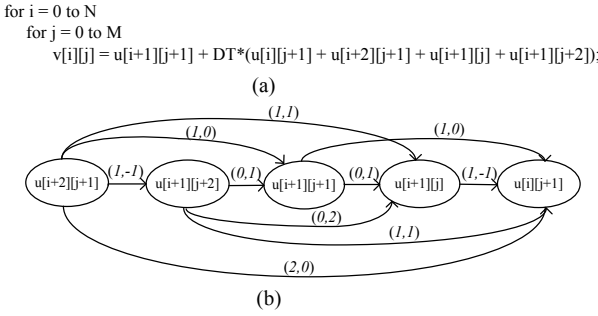

Figure 3: (a) Normalized kernel loop of rician-denoise. (b) Reuse candidate graph built on (a).

(V_s, $V_d \in V_G$) in E_G represents the data dependence between references V_s and V_d with reuse distance vector $\vec{d}(V_s,V_d)$. Assume $\vec{d}(V_s,V_d) = (d_1,d_2, \ldots, d_n)$, the length of reuse edge $V_s \rightarrow V_d$, denoted by $l(V_s,V_d)$, is defined to be $\sum_{i=n}^{1}(d_i\prod_{j=i}^{n}U_{j+1})$ where U_j is the upperbound of j^{th}-level loop nest ($U_{n+1} = 1$).

One example of a reuse candidate graph built for the kernel code in the *rician-denoise* [17] application is shown in Figure 3. Each vertex in the reuse candidate graph represents one array reference. The directed edge from $u[i+1][j+1]$ to $u[i][j+1]$ with reuse distance vector $\vec{d} = (1,0)$ implies that $u[i][j+1]$ at iteration $\vec{k}+\vec{d}$ will reuse the array element accessed by $u[i+1][j+1]$ at iteration $\vec{k}$, and the length of $V_i \rightarrow V_j$ equals M. If SPM size is large enough to hold data until the next access at $\vec{d}(V_i,V_j)$ iterations later, the corresponding reuse edge $V_i \rightarrow V_j$ will be marked as an ***active*** edge. Notice that the reuse candidate graph is constructed for UGS references; a loop may have more than one reuse candidate graphs. The reuse candidate graph of irregular or non-affine references only contains one vertex, namely the reference itself.

In order to analyze reuse possibility and calculate the number of required data transfers into SPM, *local region* and *reuse region* are defined for each vertex in the reuse candidate graph.

DEFINITION 3. *Given reuse candidate graph G with iteration space U, for each reference V_k in G, we define the local region of V_k to be the iteration subspace in which data accessed by reference V_k is prefetched from lower-level memory, denoted by L_{V_k}; V_k's reuse region is defined to be the iteration subspace in which access to V_k can reuse data stored in SPM for other references, denoted by R_{V_k} and $R_{V_k} = U - L_{V_k}$.*

From Definition 3 we can conclude that the total size of *local region* of all the vertices in G equals the amount of data needed to be brought into SPM, since the data accessed in the *local region* of a given memory reference is prefetched directly from lower-level memory instead of reusing its parent memory reference in G.

Based on the discussion above, we can formulate the *reuse − aware SPM prefetching problem* as follows:

Given the reuse candidate graphs $\mathbb{G}$ constructed for a loop nest, the maximal SPM size S and the estimated prefetch latency P, select a set of reuse edges to be active and create a SPM buffer for each vertex in G accordingly to hide the access latency P, so that the number of required data transfers from conventional memory hierarchy is minimized, under the constraint that the total size of the allocated SPM buffers cannot exceed S.

3.2 SPM Buffer Allocation

In our proposed scheme, one single SPM is seen as a one-dimensional address space and shared among all the inner-loop memory references. In previous work, affine address transformation has been used to map original data addresses to the corresponding address in SPM [18], [19]. The transformed SPM address space is not compact, which will lead to a waste of the limited SPM memory resource.

In the proposed management scheme, each vertex V_k in the reuse candidate graph will be allocated a SPM buffer buf_{V_k} of size L. In order to hide the memory access latency, L has to be larger than the estimated prefetch latency P, as discussed in Section 2. In this case array access $A[i]$ in a normalized loop nest will be mapped to an SPM address at $SPM[pos_A + i\%L]$, where SPM represents the entire one-dimensional SPM memory space and pos_A is the starting address of A's buffer.

Given reuse candidate graph G and a set of selected active reuse edges E, the SPM buffer size L allocated for each reference V_k in G is set as follows:

$$L = min\{P + 1 + l(V_k,V_m), |L_{V_k}|\} \quad (1)$$

In Equation 1, $|L_{V_k}|$ is the size of V_k's local region and $V_k \rightarrow V_m$ represents the longest active outgoing edge of V_k. Equation 1 can be derived from the following two cases:

Case 1 : There is no active outgoing edge of V_k in E, namely $l(V_k,V_m)$ equals 0. In this case, V_k will not be reused by any other vertex, hence SPM is merely used as a prefetch buffer of size $P+1$. However, if the amount of data needed to be brought into SPM, namely $|L_{V_k}|$, is smaller than $P+1$, SPM buffer size is set to $|L_{V_k}|$.

Case 2 : There exist active outgoing edges of V_k in E, which means V_k will be reused later by other vertices. If $|L_{V_k}| \geq P + 1 + l(V_k,V_m)$, data accessed at iteration $\vec{i}$ of vertex V_k will be stored in SPM until its next access at iteration $\vec{i} + \vec{d}(V_k,V_m)$ and be replaced with data prefetched at iteration $\vec{i} + \vec{d}(V_k,V_m)$. Hence the required SPM buffer size equals $P + 1 + |\vec{d}(V_k,V_m)|$, namely $P + 1 + l(V_k,V_m)$; Otherwise, only data in the local region of V_k need to be prefetched into SPM, the allocated SPM buffer size equals $|L_{V_k}|$.

As shown in Figure 2(c), the reuse distance vector between $fmap[i-4]$ and $fmap[i]$ is (4); therefore the SPM buffer size for $fmap[i]$ equals $P+5$. Buffer size for $fmap[i-4]$ is set to 4 (assume $P \geq 4$) since the size of its local region equals 4.

3.3 Data Transfer Measurement

THEOREM 1. *Assume that V_i in reuse candidate graph G has no active incoming edges, the number of reduced data transfers by*

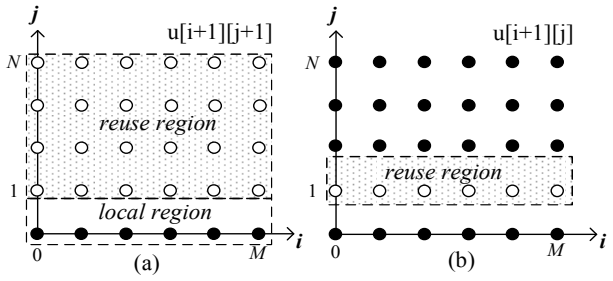

Figure 4: (a) Iteration space partition of reference u[i+1][j+1]. (b) Iteration space partition of reference u[i+1][j].

activating reuse edge $V_i \rightarrow V_j$ with reuse distance vector $\vec{d}(V_i,V_j)$ = $(d_1, d_2, \ldots, d_n)$, equals $\prod_{k=1}^{n}(U_k - |d_k|)$.

PROOF. Two conditions need to be satisfied to ensure that memory reference V_j at iteration $(t_1,t_2,\ldots,t_n)$ can reuse V_i at $(d_1,d_2,\ldots,d_n)$ iterations before: (1) $0 \le t_k \le U_k$, $\forall\, k \in [1, n]$ (U_i is the upperbound of the k^{th}-level loop); (2) $0 \le t_k - d_k \le U_k$, $\forall\, k \in [1, n]$. The two conditions are derived from the fact that both the first and second accesses fall into the iteration space. The number of iterations satisfying (1) and (2) is the total number of reuses that occur. □

Theorem 1 can be used to calculate the number of remaining data transfers given a set of active reuse edges. However, in Figure 3(b), suppose edge $u[i+1][j+2] \rightarrow u[i+1][j+1]$ and edge $u[i+1][j+1] \rightarrow u[i+1][j]$ are both selected as active edges, Theorem 1 still works for vertex $u[i+1][j+1]$ since $u[i+1][j+2]$ has no active incoming edge, while it is not the case for vertex $u[i+1][j]$. The iteration subspace R, in which access to vertex $u[i+1][j+1]$ can reuse earlier $u[i+1][j+2]$, is shown in Figure 4(a) with soft dots. Vertex $u[i+1][j]$ at iteration $\vec{t}$ will reuse $u[i+1][j+1]$ at iteration $\vec{t}-1$. If iteration $\vec{t}-1$ locates in region R, $u[i+1][j]$ needs to go upwards to visit vertex $u[i+1][j+2]$ at iteration $\vec{t}-2$. However, the allocated SPM buffer for vertex $u[i+1][j+2]$ is only $P+2$ which only can hold data of one more iteration; hence the reuse attempt of $u[i+1][j]$ will fail in this case.

For a vertex with active incoming edges, the size of its local region equals the number of required data transfers into SPM. Figure 4(b) shows the local and reuse region of $u[i+1][j]$ where reuse along edge $u[i+1][j+1] \rightarrow u[i+1][j]$ is enabled. In general, the local and reuse region of vertex V_k can be derived as follows:

THEOREM 2. *Given reuse candidate graph G with iteration space U, assume the active incoming edge set of vertex V_k is $\{V_{i_1} \rightarrow V_k, V_{i_2} \rightarrow V_k, \ldots, V_{i_n} \rightarrow V_k\}$, V_k's reuse region $R_{V_k} = \{t \mid t \in U \wedge ((t - d(V_{i_1},V_k) \in L_{V_{i_1}} \vee t - d(V_{i_2},V_k) \in L_{V_{i_2}} \ldots \vee t - d(V_{i_n},V_k) \in L_{V_{i_n}})\}$; V_k's local region $L_{V_k} = U - R_{V_k}$. (proof is omitted due to page limit)*

Theorem 2 can be applied to vertices of a given reuse candidate graph in topological order to identify their local and reuse regions, i.e., starting from the root vertex which has no active incoming edge and its local region is the entire iteration space U.

3.4 Reuse-Aware SPM Prefetching Algorithm

In this section we present a reuse-aware SPM prefetching algorithm, namely *RASP*, aimed at hiding memory access latency and minimizing data transfers from lower-level memory. In general, a SPM buffer is allocated to each vertex in the reuse candidate graph, either for pure prefetching or unified prefetching and reuse.

Algorithm 1 Reuse-Aware SPM Prefetching (RASP) Algorithm

```
1: U      → iteration space of loop nest l
2: 𝔾      → reuse candidate graph set constructed for loop nest l
3: 𝔼      → set of activated edges in 𝔾
4: S      → maximal size of SPM storage
5: P      → estimated prefetch latency
6:
7: For all the vertices v in 𝔾, set initial SPM buffer size to be P + 1 with
   local region U
8: while 1 do
9:    traverse all the unactivated edges in 𝔾 and calculate their SPM uti-
      lization ratio
10:   activate the edge (u,v) with largest positive utilization ratio under
      SPM size constraint
11:   add edge (u,v) to 𝔼
12:   add all the edges (u,v') to 𝔼 if l(u,v') ≤ l(u,v)
13:   update SPM buffer size of u
14:   update local/reuse region of v, {v'} and vertices reachable from v
      and {v'} along edges in 𝔼
15:   if size of 𝔼 remains the same then
16:      break;
17:   end if
18: end while
```

To find the active reuse edge sets with minimum required data transfers under SPM size constraint is NP-hard, as one can reduce a Knapsack problem to it (proof is omitted due to page limit). To balance the runtime overhead, we propose a heuristic algorithm to approximate the optimal solution.

As shown in the RASP algorithm, edges in the reuse candidate graphs are activated one by one under the maximal SPM size constraint. Here *activate* edge $u \rightarrow v$ means allocate a SPM buffer for u which is large enough for the corresponding reuse to occur. Hence, when edge $u \rightarrow v$ is activated, all the edges starting at u with a smaller required SPM size, namely a smaller edge length, should also be activated accordingly, as shown in line 12.

The metric used to indicate SPM utilization efficiency is called *SPM utilization ratio*, which equals the ratio of data transfer reduction to the buffer size increment by activating a given reuse edge (u,v). The amount of reduced data transfers after activating edge (u,v) equals the size difference of the local region of vertex v and all the vertices reachable from v along selected active edges, which can be obtained with Theorem 2.

Lines 10-13 show that after an edge ending at vertex v has been activated, the local/reuse region updates are applied to vertex v and the vertices reachable from v along edges in the current active set $\mathbb{E}$. The updates for v's downstream vertices are necessary since the reduction of v's local region after activating edge (u,v) has further impact to v's descendants.

THEOREM 3. *The worst-case time complexity of the RASP algorithm is $O(n^4)$, where n is the number of vertices in the reuse candidate graph set. (proof is omitted due to page limit)*

After the finalization of the activated edge set, the prefetching scheme for each vertex can be determined accordingly. v will be removed from the prefetching set for iterations in its reuse region.

4. EXPERIMENTAL RESULTS

We evaluate the proposed RASP scheme on a simulation platform built upon Simics [20] with GEMS [21]. Omega library [22] is used for memory reuse analysis. The energy results are obtained with HP McPAT tool [23]. Table 1 shows the architecture parameters used in our model.

The first-level memory is partitioned into programmer-transparent cache and compiler-managed SPM memory space at

Table 1: Simulation parameters.

Core	Sun UltraSPARC-III Cu processor core
L1 Memory	32KB, 64-byte block, 2-cycle access latency
L2 Cache	512KB, 64-byte block, 20-cycle access latency
Main Memory	4GB, 320-cycle access latency

the ratio of 1:3, 2:2 or 3:1, which is close to Fermi's 3:1 and 1:3 configuration points. The cache-to-SPM ratio customized to each benchmark is shown in Table 2, which is obtained from offline profiling. An in-flight counter is added for each SPM block to check whether prefetching has finished or not. This can ensure the correctness of program functionality.

Our test cases include six benchmarks from the *SPEC*2006 suite ($bzip2$, mcf, $hmmer$, $libquantum$, $h264ref$ and lbm) and two applications from the medical imaging area ($mutualInfo$ and *rician-denoise*). These benchmarks are chosen as they have intensive memory references. For the two medical imaging applications, $mutualInfo$ computes the mutual information of two 2D images and *rician-denoise* performs iterative local denoising based on *rician-noise* model. The runtime overhead of RASP algorithm on each test case is less than 10 seconds.

The proposed RASP scheme is compared with three reference points in our experiments. The first reference point is cache prefetching with the entire L1 memory allocated to conventional cache, and we have enabled current state-of-the-art commercial prefetching scheme offered in SUN's CC compiler [24]. The second reference point is a hybrid memory system in which SPM is used as a prefetch buffer, following the prefetch-only scheme discussed in Section 2.1. The third reference point is the DRDU-generated SPM management scheme [1]. The cache-to-SPM ratio adopted for the second and third reference points also follows Table 2.

4.1 Comparison Results

Performance. Figure 5 shows the overall performance comparison results for the eight benchmark kernels, where the four bars correspond to the cache prefetching scheme, SPM prefetch-only scheme, DRDU and RASP scheme. As shown in the figure, RASP exhibits speedup ranging from 9.6% to 44.3% in six out of eight applications, compared to the cache-only scheme. In *hmmer* a slight performance degradation occurs. The reason is that the access pattern in *hmmer* has strong data locality which can be captured well by conventional cache architecture. In addition, cache pollution is less likely to happen here since the next access will occur soon. In this case the extra instruction overhead of calculating the SPM address cannot be offset by the small amount of reduced data transfers.

When compared to the SPM prefetch-only scheme, mcf and lbm are two special cases in which data access patterns are either random or streaming. In this case the generated SPM prefetch-only scheme is exactly the same as RASP. This also explains the small performance difference, when compared to DRDU in these two applications. For most of the remaining applications, the performance improvement over the SPM prefetch-only scheme is less than that over cache, since cache pollution is avoided in the SPM prefetching scheme.

On average RASP has achieved a 15.9%, 12.9% and 18.5% performance improvement over cache prefetching, prefetch-only SPM management and DRDU results. The corresponding maximal gains are 44.3%, 39.3% and 32.8%, respectively.

Memory Access Latency. The comparison of normalized memory access latency is shown in Figure 6. We can see that the

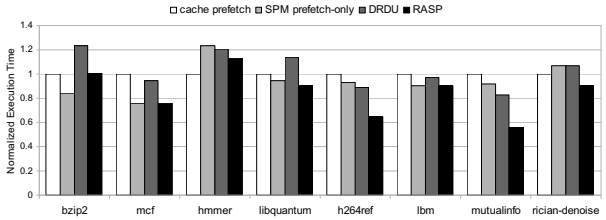

Figure 5: Comparison of execution time.

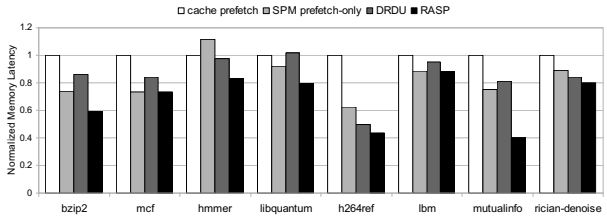

Figure 6: Comparison of memory access latency.

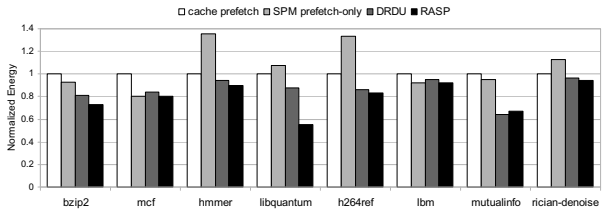

Figure 7: Comparison of energy consumption.

memory access latency reduction in RASP is larger than the performance improvement for most of the test cases, when compared to cache prefetch and the SPM prefetch-only scheme. This can be explained by the instruction overhead introduced by explicit SPM management. In summary, RASP has shown an average 31.6% memory access latency reduction over the cache-only case, a 26.4% reduction over the SPM prefetch-only case and an average 19.5% reduction over DRDU result. The corresponding maximal gains are 59.6%, 46.2% and 50.3%, respectively.

Energy Consumption. Figure 7 shows the energy consumption comparison among the four schemes. Since cache can take advantage of the existing data locality in the program and save further access to lower-level memory, a 6% energy decrease of cache prefetching over the SPM prefetch-only scheme is observed.

On the other hand, up to 44.7%, 42.6% and 27.7% energy gains are achieved by RASP over the other three schemes. The reasons include the intrinsic less energy consumption for SPM access as well as the reduced number of accesses to lower-level memory by efficiently utilizing the reuse pattern with SPM. The average energy consumption reduction of RASP is 22% and 31.2% over cache and the SPM prefetch-only case, respectively. The average 10% energy reduction over DRDU comes from the improved execution time, as well as the reduced SPM data transfers.

4.2 Discussion of SPM Utilization Efficiency

Figure 8 shows the comparison between RASP and DRDU in terms of SPM buffer size and the number of data transfers from lower-level memory. We only include applications with regular

Table 2: L1 memory configuration.

	bzip2	mcf	hmmer	libquantum	h264ref	lbm	mutualInfo	rician-denoise
SPM	8KB	8KB	8KB	24KB	8KB	16KB	24KB	16KB
L1 Cache	24KB	24KB	24KB	8KB	24KB	16KB	8KB	16KB
SPM vs. Cache	1:3	1:3	1:3	3:1	1:3	2:2	3:1	2:2

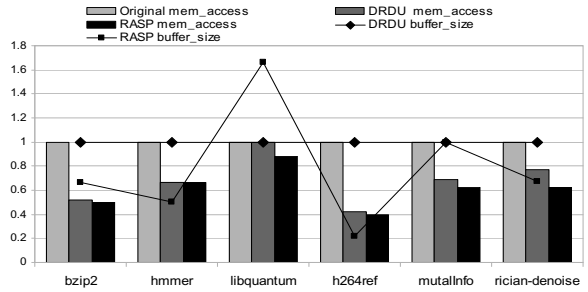

Figure 8: Comparison of buffer size and SPM data transfers.

reuse patterns in this comparison. The same SPM size constraint is applied to the two approaches and the prefetching scheme in RASP is disabled for fairness. In general, we can see that the number of data transfers from lower-level memory of RASP is 7% smaller over the DRDU result and 41.2% smaller over the original program. The buffer size of RASP is 22.7% smaller than the DRDU buffer size. The smaller required SPM size in RASP scheme implies a higher SPM storage utilization ratio, which also provides more space to harmonize with SPM management techniques working on other program elements, e.g. [25].

5. CONCLUSION

In this work we introduce a reuse-aware SPM prefetching scheme to efficiently utilize SPM memory space. The proposed SPM prefetch scheme shows a significant performance/power improvement against previous SPM management techniques, which demonstrates the impact of reuse patterns on SPM prefetching efficiency. Note that the proposed scheme can be combined with traditional techniques of data locality optimization, e.g., loop interchange or tiling, to further improve the usage of SPM. The co-optimization effectiveness will be investigated in our future work.

6. ACKNOWLEDGMENTS

This work is partially supported by MARCO Gigascale Systems Research Center (GSRC), the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, and the NSF grant CCF-0903541. We also would like to give special thanks to Ilya Issenin and Prof. Nikil Dutt in University of California, Irvine for DRDU source code sharing.

7. REFERENCES

[1] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "DRDU: A Data Reuse Analysis Technique for Efficient Scratch-Pad Memory Management," in *ACM Trans. Des. Autom. Electron. Syst.*, 2007.

[2] T. Chen, T. Zhang, Z. Sura, and M. Tallada, "Prefetching Irregular References for Software Cache on Cell," in *Proc. CGO*, 2008, pp. 155–164.

[3] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," in *Proc. CODES*, 2002, pp. 73–78.

[4] J. Sjodin and C. Platen, "Storage Allocation for Embedded Processors," in *Proc. CASES*, 2001, pp. 15–23.

[5] O. Avissar, R.Barua, and D.Stewart, "An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems," in *ACM TRANS. Embed. Comput. Syst.*, 2002, pp. 6–26.

[6] M. Verma, S.Steinke, and P. Marwedel, "Data Partitioning for Maximal Scratchpad Usage," in *Proc. ASPDAC*, 2003, pp. 77–83.

[7] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic Management of Scratchpad Memory Space," in *Proc. DAC*, 2001, pp. 690–695.

[8] S. Udayakumaran and R. Barua, "Compiler-decided Dynamic Memory Allocation for Scratchpad Based Embedded Systems," in *Proc. CASES*, 2003, pp. 276–286.

[9] L. Li, H. Feng, and J. Xue, "Compiler-directed Scratchpad Memory Management via Graph Coloring," in *ACM Trans. Archit. Code Optim.*, 2009, pp. 1–17.

[10] T. Yemliha, S. Srikantaiah, M. Kandemir, and O. Ozturk, "SPM Management Using Markov Chain Based Data Access Prediction," in *Proc. ICCAD*, 2008, pp. 565–569.

[11] A. Beric, R. Sethuraman, H. Peters, G. Veldman, J. Meerbergen, and G. Haan, "Streaming Scratchpad Memory Organization for Video Applications," in *Proc. Circuits, Signals and Systems*, 2004, pp. 427–432.

[12] T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in *Proc. ASPLOS*, 1992, pp. 62–73.

[13] S. Vanderwiel and D. Lilja, "Data Prefetch Mechanisms," in *ACM Computing Surveys*, 2000, pp. 174–199.

[14] R. M. Rabbah, H. Sandanagobalane, M. Ekapanyapong, and W. Wong, "Compiler Orchestrated Prefetching via Speculation and Predication," in *Proc. ASPLOS*, 2004, pp. 189–198.

[15] T. C. Mowry, "Tolerating latency through software-controlled data prefetching," Ph.D. dissertation, Stanford University, 1994.

[16] K. Kennedy and J. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[17] *ITK Software Guide*, http://www.itk.org/ItkSoftwareGuide.pdf.

[18] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Bountev, and P. Sadayappan, "Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories," in *Proc. PPoPP*, 2008, pp. 1–10.

[19] M. Kandemir and A. Choudhary, "Compiler-Directed Scratchpad Memory Hierarchy Design and Management," in *Proc. DAC*, 2002, pp. 628–633.

[20] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," in *IEEE Computer*, 2002, pp. 50–58.

[21] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator(GEMS) toolset," in *Computer Architecture News*, 2005, pp. 92–99.

[22] *Omega Library*, http://www.cs.umd.edu/projects/omega.

[23] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-core and Many-core Architectures," in *Proc. MICRO*, 2009, pp. 469–480.

[24] Sun Microsystems, "UltraSPARC-II Enhancements: Support for Software Controlled Prefetch," White Paper, 1997.

[25] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU," in *IEEE Trans. on Computers*, vol. 59, no. 8, 2010, pp. 1047–1062.