Matched SAMS Scheme: Supporting Multiple Stride Unaligned Vector Accesses with Multiple Memory Modules

Chunyang Gou, Georgi Kuzmanov and Georgi N. Gaydadjiev
Computer Engineering Lab
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
{C.Gou, G.K.Kuzmanov, G.N.Gaydadjiev}@tudelft.nl

Abstract

In this paper, we analyze the problem of supporting conflict-free access for multiple stride families in parallel memory schemes targeted for high performance vector processing systems. We propose the Matched SAMS Scheme, which is based on the basic SAMS scheme, to support conflict-free vector memory accesses for strides from multiple stride families. We compare our scheme against previously proposed techniques, e.g. using buffers and inter-vector out-of-order access. The main advantage of our proposal is that the atomic parallel access is supported without limiting the vector lengths. This provides better support for short vectors. Our scheme also has the merit of better memory module utilization compared to the solutions with additional modules. Synthesis results for TSMC 90 *nm* Low-K CMOS technology indicate that the Matched SAMS Scheme has efficient hardware implementations, with a critical path delay of less than 1 *ns* and moderate hardware resource utilization. To validate the performance of proposed Matched SAMS Scheme in real applications, we applied it to IBM Cell processor by integrating it to the Cell SPE local store, and did experiments with applications from PARSEC benchmarks and micro-kernels from IBM Cell SDK. Simulation results show that with the direct support of unaligned and strided memory access patterns by our parallel memory scheme, the dynamic instruction counts drops by up to 49%, which turns into a reduction of around 46% in execution time.

1 Introduction

One of the most critical design challenges in SIMD processors is imposed by the memory subsystem, which is required to deliver sustained high bandwidth at reasonable latency [13, 23]. To meet these challenges, memory subsystems with multiple memory modules have been widely considered. Parallel (or multimodule) memories were introduced in the early years of building high performance processors [8] and later extensively adopted in vector supercomputers [32, 17]. Nowadays, there is a trend that general purpose systems are utilizing parallel memories in their memory hierarchy, such as the multibank on-chip caches in Niagara [21] and Opteron [20], multislice caches in Power processors [35, 33, 24], parallel on-chip eDRAM banks in VIRAM processor [23], and interleaved DRAM banks in Rambus and other commercial-off-the-shelf monolithic DRAM chips. For simplicity of the module assignment hardware implementation, the number of memory modules was chosen as a power of two in most of these products. For efficient hardware utilization, the designers prefer systems with non-redundant memory schemes, i.e. schemes where each and every memory module can be referenced by any memory access. This paper addresses non-redundant memory systems with a power of two memory modules. Based on SAMS [16], we introduce the Matched SAMS Scheme, which overcomes the problem of conflict-free[1] access across stride families[2] in multimodule parallel memory systems. The specific contributions of our proposal are:

[1]Please refer to Section 2.1 for detailed explanation of "conflict-free access".
[2]See the definitions in Section 2.3.

- We propose the Matched SAMS scheme, which support conflict-free vector accesses with strides from multiple stride families. Moreover, the number of supported stride families is increased from 2 to $log_2(\#modules) + 1$, compared to the original SAMS scheme;
- We present the mathematical foundations for both the SAMS and Matched SAMS schemes;
- We have implemented the entire Matched SAMS memory system, and synthesis results on TSMC 90 nm Low-K CMOS technology suggest short critical paths (less than 1 ns). This is a strong indication for the feasibility of the proposed scheme in practical parallel memory systems.
- We have integrated the Matched SAMS Scheme into IBM Cell SPE loclal store memory and investigated its performance on several real applications from PARSEC benchmark and micro-kernels from IBM Cell SDK. Simulation results show that with the direct support of unaligned and strided memory access patterns by our parallel memory scheme, the dynamic instruction counts drops by up to 49% on average, which turns into a reduction of around 46% in execution time.

The remainder of the paper is organized as follows. In Section 2, we present the background and motivation of this work. In Section 3, the mathematical equations of the proposed Matched SAMS Scheme are described, followed by the mathematical validation of the SAMS scheme and its derivative, the Matched SAMS Scheme in Section 4. The hardware implementation and synthesis results of the Matched SAMS Scheme and discussions about the hardware implementation variants are presented in Section 5. The integration of the Matched SAMS Scheme into IBM Cell processor and its simulated performance in several applications and kernels are investigated in Section 6. The major differences between our proposal and related art are described in Section 7. Finally we conclude the paper in Section 8.

2 Background and Motivation

In this section, we will introduce some background on parallel memory schemes, which play a central role in parallel memory systems. We also present some of the key existing techniques in parallel memory schemes. Then we will present the limitation in nonredundant parallel memory schemes, which motivates us to this work.

2.1 Parallel Memory Schemes

The parallel memory schemes are the main means to determine the performance and the hardware complexity of the parallel memory subsystems. Given a specific physical memory organization and resources, these schemes determine the mapping from the linear address space to the physical location identifiers, such as the module number and row address. In other words, the memory translation scheme determines how to distribute data to different memory banks, in order to better service memory references. Vector access, defined by an address stream with a constant offset between any two consecutive addresses, is one of the most important memory reference patterns in SIMD applications. Traditional parallel memory schemes in vector computers provide conflict-free access for a *single* stride family. To solve the module conflicts encountered with the cross stride family accesses, several enhancements have been previously proposed, including the use of dynamic memory schemes [11, 10], use of buffers [9], and use of more memory modules[3] (i.e. memory scheme with redundant modules) [9], and out-of-order vector access [36].

These traditional multimodule memory schemes assume that the memory access time is atomic and it is much larger than the processor cycle time. More importantly, to keep up with the data access demand of the fast processors, different memory schemes were proposed to make multiple memory modules work

[3]Note we will refer to "use of more memory modules" as "use of redundant memory modules" or "redundant memory schemes" in this paper. Keep in mind that there is actually no data duplication in redundant memory shcemes.

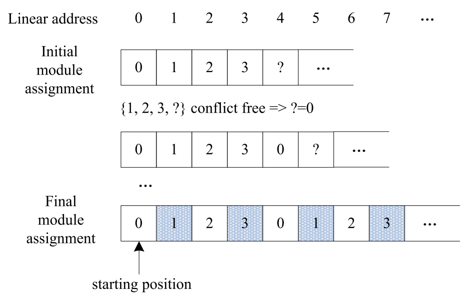


Figure 1: Inherent limitation in multimodule memory assignment

together in an *interleaved* manner to service single cycle data accesses from the processor, in traditional parallel memory schemes. Particularly, if the number of memory modules equals to their latency counted in processor cycles, the scheme is called a *matched* memory system, as in this case, the interleaved memory modules are capable of servicing the processor at the throughput of one datum per processor cycle, if the data to be referenced in one access are located in different memory modules. This condition of even distribution of memory references is called **conflict-free** access in traditional interleaving schemes. In conflict-free access, data could be accessed in parallel as there is no module conflict during the memory access.

With the advance of the semi-conductor technologies, however, the parallel memory modules of traditional vector computers could be (partially) integrated into the processor chip working as a local store like the Cell processor [19]. Thus, on the one hand, for medium-sized on-chip memory arrays (such as the L1 caches in GPPs), it is not difficult to support one access per processor cycle. On the other hand, to exploit the available on-chip resources more efficiently, multiple SIMD data paths are deployed in modern processors. Furthermore, accessing multiple memory data items in a single processor cycle is desirable in order to use the multiple data paths efficiently. Therefore, we assume a memory organization where all memory modules are on chip[4] and each of them services one piece of data reference every clock cycle. In this work, we address such memory organizations and refer to them as "*fully-parallel* multimodule memory systems". In fully-parallel multimodule memory systems, the processor could access as many data items as the number of memory modules in any cycle, when the access is conflict-free. This is actually the goal of the underlining parallel memory schemes in such memory organizations.

2.2 Motivation: The Limitation in Nonredundant Parallel Memory Schemes

In traditional matched parallel memory schemes, it is impossible to simultaneously support parallel strided vector accesses with strides from multiple stride families [36]. Figure 1 illustrates an example with four memory modules. Under the constraint of unit-stride conflict-free access, the module assignment function of the scheme is completely fixed. Note in Figure 1 the constant repeat of module assignment pattern of the first four addresses. When the system is accessed with stride 2, half of the memory modules are not utilized (indicated by the shadowed cells in Figure 1).

There is a large number of strided vector accesses in many scientific and engineering applications which have significant impact on the performance of the workloads on traditional vector supercomputers [6]. In the meanwhile, we certainly could not neglect the unit-stride access pattern, as it is the most common one in vectorized scientific and engineering applications [18, 22, 34]. Even in vectorized SPEC95 benchmarks it is the second most frequent stride [28]. Furthermore, there are many occasions in which simultaneous support of vector memory accesses with strides from multiple stride families is desired, as the same data block is accessed with different vector patterns. When we have to access data in parallel memories with multiple strides, the problem occurs that we have to either modify the interleaving scheme (that is, to redistribute data to memory modules in a different way), or to have the scheme optimized for conflict-free access with one type of access while suffer from the non-conflict-free access with the other. The former would incur data

[4]Three are already many GPPs with multimodule on-chip SRAMs, see [24, 19, 21] for examples.

flushing into and reloading from the lower level memory in the memory hierarchy whenever there is a change of access stride, whereas the latter would incur additional processor cycles waiting for the vector access.

2.3 Definitions

For the sake of clarity, we now give the definitions of some terminologies used in this paper.

Definition 1. A sequence of independent memory access stream issued by the SIMD processor in parallel is called **vector access**. Vector access could be either regular (with constant stride) or irregular (such as the scatter/gather memory access), however we only discuss regular vector access in this paper.

Definition 2. **Base address** is the first memory address in a given regular vector access stream.

Definition 3. **Stride** is the constant interval between subsequent memory addresses in a given regular vector access stream.

Definition 4. **Unit stride** denotes stride 1.

Definition 5. A **stride family** is a set of infinite number of strides, $\{S \| S = \sigma \cdot 2^s,\ s \in \mathbb{N},\ \sigma \text{ is odd}\}$. This follows the definitions given in [11, 10, 36].

Definition 6. The exponential part of the stride family $\{S \| S = \sigma \cdot 2^s,\ s \in \mathbb{N},\ \sigma \text{ is odd}\}$, s, is called the **stride family number**. The stride family number completely defines the set of strides belonging to the stride family. For example, stride family 0 is the stride set $\{1, 3, 5, 7, \cdots\}$ while stride family 1 is $\{2, 6, 10, 14, \cdots\}$.

3 The Matched SAMS Scheme

In this section, we introduce the Matched SAMS Scheme, which is derived from SAMS, the Single-Affiliation Multiple-Stride conflict-free parallel memory scheme. SAMS was initially proposed to simultaneously support conflict-free unit-stride and strided memory accesses from one *single* stride family in [16], by first constructing a single-affiliation interleaving scheme, and then making data lines wider to solve the module conflicting problem in unit-stride access. The Matched SAMS Scheme takes a step further in that it supports conflict-free vector accesses for strides from *multiple* stride families.

3.1 SAMS Parallel Memory Scheme

As described in [16], the SAMS scheme consists of three functions: (1) the module assignment function which assigns an item in linear address space to a specific module; (2) the row assignment function which determines the row in which the item is placed; and (3) the offset assignment function which calculates the offset of the item in the row, as listed in the following:

- *module assignment function:*

$$m(a) = \begin{cases} a\%2^q, & s = 0 \\ \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,\,q+1}}\right)\%2^{s-1}\right\rangle, & 1 \leq s \leq q \\ \left(a \otimes T_{H_{q,s}}\right)\%2^q, & s > q \end{cases} \qquad (s,\, q \in \mathbb{N})$$

- *row assignment function:*

$$r(a) = \begin{cases} \frac{a}{2^{q+1}}, & s = 0 \\ \frac{a}{2^{q+1}}, & 1 \leq s \leq q \\ \left(\left(\frac{a}{2^q} + 1\right)\%2^{n-q}\right)/2, & s > q \end{cases} \qquad (s,\, q \in \mathbb{N})$$

- *offset assignment function:*

$$o(a) = \begin{cases} a_q, & s = 0 \\ a_{s-1}, & 1 \leq s \leq q \\ \overline{a_q}, & s > q \end{cases} \qquad (s,\, q \in \mathbb{N})$$

where, a is the n bit linear address, which is implemented by the 2^q memory modules in the SAMS scheme; s is the *stride family number* to be supported with conflict-free access by the scheme; a_i is the *i-th* bit of a. The notation $x\%y$ means x modulo y, and Notations x/y and $\frac{x}{y}$ mean the quotient of integer division between x and y. $< \ldots,\ \cdots >$ denotes binary bits concatenation. $T_{H_{x,y}}$ is the XOR scheme address transformation matrix taken from [10]. $T_{H_{x,y}} = \prod_{k=0}^{\min(x,y)-1} T_{k+\max(x,y),\,k}$, where $T_{i,j}$ is defined to be the identity matrix with a single off-diagonal 1 in $T(i,j)$. The binary matrix T is arranged in a form such that the bottom-right element is $T(0,0)$, and the row index grows when moving up and the column index grows when moving left so that the top-left element is $T(l-1, l-1)$ (assume the size of T is $l \times l$). For example,

$$T = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = T_{1,0} \cdot T_{2,1}$$

The $\otimes$ symbol in this paper is used for binary vector-matrix multiplication. For instance, consider

$$a = 7,\ T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$a \otimes T = [1\ 1\ 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = [1\ 1\ 0] = 6\ .$$

3.2 The Matched SAMS Scheme

In this paper, we define the special case of SAMS scheme under the condition $s = q$ as **Matched SAMS Scheme**. In the Matched SAMS Scheme, the system parameter s is fixed to q, thus the module assignment function, the row assignment function and offset assignment function are simplified as follows:

$$\begin{cases} m(a) &= \left\langle a_q,\ \left(a \otimes T_{H_{q-1,\,q+1}}\right)\%2^{q-1}\right\rangle \\ r(a) &= \frac{a}{2^{q+1}} \\ o(a) &= a_{q-1} \end{cases} \qquad (q \in \mathbb{N}) \qquad (1)$$

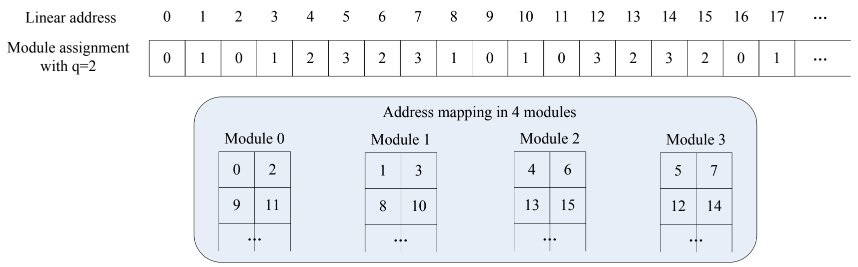


Figure 2: Address mapping in the Matched SAMS Scheme with 4 modules

Let's consider $q = 2$, which means the Matched SAMS Scheme with 4 memory modules, for example. With $q = 2$ we have

$$\begin{aligned} m(a) &= \left\langle a_2,\ \left(a \otimes T_{H_{1,\,3}}\right)\%2\right\rangle \\ &= \left\langle a_2,\ \left(a \otimes T_{3,\,0}\right)\%2\right\rangle \\ &= \left\langle a_2,\ a_3 \oplus a_0\right\rangle \end{aligned}$$

. Therefore, the address mapping of the Matched SAMS Scheme with 4 memory moduels is

$$\begin{cases} m(a) &= \langle a_2,\ a_3 \oplus a_0\rangle \\ r(a) &= a_{n-1:3} \\ o(a) &= a_1 \end{cases}$$

. Figure 2 illustrates the address mapping of the above example. Taking the base address 1 for example, the referenced linear address groups for stride 1, 2 and 4 vector accesses are {1, 2, 3, 4}, {1, 3, 5, 7} and {1, 5, 9, 13}, respectively. As Figure 2 shows, all addresses in each group could be accessed in parallel within the Matched SAMS Scheme. Thus the Matched SAMS Scheme is capable of supporting conflict-free vector access with strides from more than 2 stride families, which will be proved in next section.

4 Proof of Conflict-Free Vector Access

In this section, we will present the mathematical fundations of the basic SAMS scheme, and its derivative, the Matched SAMS Scheme. As already discussed in Section 2.1, "conflict-free" conventionally means that data to be referenced in one access are located in different modules so that they could be accessed in parallel. However the concept of conflict-free in this paper is slightly extended such that it includes the cases whenever there are more than one references located in the same module whereas they could also be accessed in parallel as they are in the same row. For the sake of clarity, we will use the term "strictly conflict-free" when we refer to the conventional meaning. Now we will first illustrate some properties of the SAMS scheme, and then give the proof of its capability of supporting conflict-free access in theorems later.

Property 1. The period of SAMS module assignment function is 2^{q+s}.

Proof. There are three cases in the SAMS scheme.

I) $s = 0$. $m(a) = a\%2^q = \left(a + 2^{q+s}\right)\%2^q = m(a + 2^{q+s})$.

II) $1 \le s \le q$.

$$\begin{aligned} m(a) &= \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\right\rangle \\ &= \left\langle a_q \cdots a_s,\ \left(\left(a + 2^{q+s}\right) \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\right\rangle \\ &= m\left(a + 2^{q+s}\right)\ . \end{aligned} \tag{2}$$

$$\begin{array}{lccc}
 & \langle a_{n-1}\ldots a_{q+1}, & a_q\ldots a_s, \; a_{s-1}, & a_{s-2}\ldots a_0\rangle \\
+ & & \delta_{q-1} \quad \ldots & \delta_0 \\
\hline
 & \langle b_{n-1}\ldots b_{q+1}, & b_q\ldots b_s, \; b_{s-1}, & b_{s-2}\ldots b_0\rangle
\end{array}$$

Figure 3: Binary bits representation of $b = a + \delta \cdot 2^0$ when $1 \leq s \leq q$

$$\begin{array}{lccc}
 & \langle a_{n-1}\ldots a_{q+1}, & a_q\ldots a_s, \; a_{s-1}, & a_{s-2}\ldots a_0\rangle \\
+ & & \delta_{q} \quad \ldots & \delta_0 \; 0 \\
\hline
 & \langle b_{n-1}\ldots b_{q+1}, & b_q\ldots b_s, \; b_{s-1}, & b_{s-2}\ldots b_0\rangle
\end{array}$$

Figure 4: Binary bits representation of $b = a + \delta \cdot 2^1$ when $1 \leq s \leq q$

Equation (2) stands because the module assignment function of Harper's XOR scheme, namely $\left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1}$, has a period of $2^{(s-1)+(q+1)} = 2^{q+s}$ [11].

III) $s > q$. In this case the module assignment function is precisely that of Harper's XOR scheme configured with 2^q banks and stride 2^s, which has a period of 2^{q+s} [11]. $\square$

Property 2. When $1 \leq s \leq q$, the SAMS scheme is conflict-free for any stride $S = 2^{s'}(0 \leq s' \leq s-1)$.

Proof. The SAMS scheme is conflict-free, if we could guarantee that the conflicting items in linear address space are mapped to the same row. Described in mathematics, given two different addresses a and b, we need prove $r(a) = r(b)$ under the conditions $m(a) = m(b)$ and $b = a + \delta \cdot 2^{s'}(1 \leq \delta \leq 2^q - 1)$(without loss of generality we assume $b > a$).

First we examine the equation $m(a) = m(b)$. Note when $1 \leq s \leq q$ $m(a) = m(b)$ means

$$\left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \right\rangle = \left\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} \right\rangle \ ,$$

i.e.

$$\begin{aligned}
a_q \cdots a_s &= b_q \cdots b_s &(3)\\
\left(a \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1} &= \left(b \otimes T_{H_{s-1,q+1}}\right) \%2^{s-1}\ . &(4)
\end{aligned}$$

According to the definition, equation (4) could be further expanded as

$$\begin{cases}
a_0 \oplus a_{q+1} &= b_0 \oplus b_{q+1} \\
a_1 \oplus a_{q+2} &= b_1 \oplus b_{q+2} \\
\cdots & \\
a_{s-2} \oplus a_{q+s-1} &= b_{s-2} \oplus b_{q+s-1}
\end{cases} . \tag{5}$$

When $s' = 0$, we have $b = a + \delta$. The binary bits representation of the addition process is depicted in Figure 3. Consider $a_q = b_q$(from equation (3)) together with Figure 3, we could see that there is no carry input from bit $q-1$ to bit q during the addition. Accordingly, the high order bits(from bit q on) of a are not affected by the addition of δ, which means $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$.

When $s' = 1$, we have $b = a + 2 \cdot \delta$, which is depicted in Figure 4. Note the addition on bit 0 is $b_0 = a_0 + 0$, thus $b_0 = a_0$. Combined with equation (5), we have $b_{q+1} = a_{q+1}$, which indicates that there is no carry input from bit q to bit $q+1$. Hence $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$.

Similarly, for $s' = k(k = 2, \dots, s-1)$, i.e. $b = a + \delta \cdot 2^k$, we have

$$\begin{cases} a_0 &= b_0 \\ a_1 &= b_1 \\ \dots \\ a_k &= b_k \end{cases} . \tag{6}$$

by examing the process of addition. Considering (6) together with (5), we know

$$\begin{cases} a_{q+1} &= b_{q+1} \\ a_{q+2} &= b_{q+2} \\ \dots \\ a_{q+k+1} &= b_{q+k+1} \end{cases} . \tag{7}$$

This indicates that there is no carry input from bit $q+k$ to bit $q+k+1$. Therefore the high order bits(from bit $q+k+1$ on) of a are kept untouched during the addition. Consequently we have $a_{n-1} \cdots a_{q+1} = b_{n-1} \cdots b_{q+1}$, i.e. $\frac{a}{2^{q+1}} = \frac{b}{2^{q+1}}$, which means $r(a) = r(b)$. □

Property 2 reveals a very interesting feature of the SAMS scheme: it could potentially support conflict-free vector accesses with strides across multiple stride families, under the condition $s \le q$. Moreover, it is exactly where the idea of the Matched SAMS Scheme originates. We will see how this feature works for the Matched SAMS Scheme later.

Before proving the conflict-free access support of SAMS scheme, first we have to prove that it is a bijection on $\mathbb{E}^n$[5]. Only when the interleaving scheme is a bijection from the linear address space to the transformed space(the module-row-offset trinity in SAMS) could it be consistent in both theory and practice.

Theorem 1. The mapping from linear address a to the module-row-offset trinity in the SAMS scheme is a bijection on $\mathbb{E}^n$.

Proof. As there are three cases in the SAMS scheme, we will discuss them one by one.

I) $s = 0$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\begin{aligned} & \langle m(a),\ r(a),\ o(a) \rangle \\ = & \left\langle a\%2^q,\ \frac{a}{2^{q+1}},\ a_q \right\rangle \\ = & \left\langle a\%2^q,\ \frac{a}{2^q} \right\rangle . \end{aligned}$$

It's clear that the mapping from a to the trinity $\langle m(a), r(a), o(a) \rangle$ is a bijection on $\mathbb{E}^n$.

II) $1 \le s \le q$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\begin{aligned} & \langle m(a),\ r(a),\ o(a) \rangle \\ = & \left\langle a_q \dots a_s,\ \left(a \otimes T_{H_{s-1,s+1}}\right) \%2^{s-1},\ \frac{a}{2^{q+1}},\ a_{s-1} \right\rangle & (8) \\ \overset{bijection}{\Leftrightarrow} & \left\langle \left(a \otimes T_{H_{s-1,s+1}}\right) \%2^{s-1},\ \frac{a}{2^{q+1}},\ a_q \dots a_s,\ a_{s-1} \right\rangle & (9) \\ = & \left\langle \left(a \otimes T_{H_{s-1,s+1}}\right) \%2^{s-1},\ \frac{a}{2^{s-1}} \right\rangle . & (10) \end{aligned}$$

Note expression (10) is virtually the Harper XOR scheme configured with 2^{s-1} memory modules and stride 2^{s+1}(the first part of the binary concatenation is the module assignment function, and the last part is the

[5]Denote $\mathbb{E}^n = \{0, 1, \dots, 2^n - 1\}$.

row assignment function). Therefore, mapping from a to (10) is a bijiection on $\mathbb{E}^n$. As the transform between (8) and (9) is also a bijiection, hence mapping from a to $\langle m(a),\ r(a),\ o(a)\rangle$ is a bijection on $\mathbb{E}^n$.

III) $s > q$. By concatenating the binary bits of the memory module assignment, row assignment, and offset assignment, we get

$$\begin{aligned}
& \langle m(a),\ r(a),\ o(a)\rangle \\
= \quad & \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)/2,\ \overline{a_q}\right\rangle \\
= \quad & \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)/2,\ \left(\left(\frac{a}{2^q}+1\right)\%2^{n-q}\right)\%2\right\rangle \\
= \quad & \left\langle \left(a \otimes T_{H_{q,s}}\right)\%2^q,\ \left(\frac{a}{2^q}+1\right)\%2^{n-q}\right\rangle & (11) \\
\overset{bijection}{\Leftrightarrow} \quad & \left\langle \left(\frac{a}{2^q}+1\right)\%2^{n-q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q\right\rangle & (12) \\
= \quad & \left[\left\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q\right\rangle + 2^q\right]\%2^n\ . & (13)
\end{aligned}$$

Note the transform between (11) and (12) is a bijection. As we know the Harper's XOR scheme which maps linear address a to the row-module concatenation $\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q\rangle$ is a bijection on $\mathbb{E}^n$, therefore the mapping from a to $\left[\langle \frac{a}{2^q},\ \left(a \otimes T_{H_{q,s}}\right)\%2^q\rangle + 2^q\right]\%2^n$ is also a bijection with fixed q. Therefore, the mapping from linear address a to $\langle m(a),\ r(a),\ o(a)\rangle$ is a bijection on $\mathbb{E}^n$. $\square$

Theorem 2. The SAMS scheme is conflict-free for both unit-stride and stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

Proof. I) $s = 0$. In this case the stride family becomes $\{S \| S = \sigma,\ \sigma\ odd\}$, which includes the unit stride. The SAMS module assignment function when $s = 0$ is the same as that of the simple low-order interleaving scheme, hence it is conflict-free for all odd stride accesses.

II) $1 \leq s \leq q$.

a) Strided access with stride $S = 2^s$.

Suppose the starting address of the strided access is b. Then the accessed items in linear address space are b, $b+2^s$, $\ldots$, $b+2^s(2^q-1)$, which is shown in Figure 5. In the figure the address sequence is rearranged into a matrix, where the address increases in a column-major manner, and each row consists of all the references to the same subgroup (Note there are 2^{q-s+1} subgroups in total.). Now we will prove each and every item in the matrix is distributed into a different memory module, by the SAMS module assignemnt function $\langle a_q \cdots a_s,\ \left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}\rangle$. By examing the address matrix we could see that item $k \cdot 2^{q+1}(k = 0, 1, \ldots 2^s-1)$ does not affect $a_q \cdots a_s$, and the only determinant factor is item $b+k\cdot 2^s(k = 0, 1, \ldots 2^{q-s+1}-1)$, which results different $a_q \cdots a_s$ for different k. In other words, the high order bits of the module assignment function is different for different rows. Now we look into the row. For the i-th row($i = 0, 1, \ldots, 2^{q-s+1}-1$), the address sequence is precisely that generated by the strided access with starting address $b + i \cdot 2^s$ and stride 2^{q+1}. Consequently, the second part of the SAMS module assignment function, which is actually the Harper's XOR scheme configured with conflict-free access for stride 2^{q+1}, designates different module indices to different address items in the same row. In other words, $\left(a \otimes T_{H_{s-1,q+1}}\right)\%2^{s-1}$ is different for each and every items in the same row. Together with the fact that $a_q \cdots a_s$ is different for different rows, we could know that each and every items in the address sequence referenced by the strided access are assigned to different memory modules. Hence the SAMS scheme is *strictly* conflict-free for access stride $S = 2^s$.

b) Strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

Since the SAMS scheme is strictly conflict-free for stride $S = 2^s$ in the sense that all referenced addresses in one access are distributed in different modules, it is also strictly conflict-free for the stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$, according to Theorem 3 in [11].

$$\begin{array}{cccc}
b & b+2^{q+1} & \cdots & b+2^{q+1}\cdot\left(2^{s-1}-1\right) \\
b+2^{s} & b+2^{s}+2^{q+1} & \cdots & b+2^{s}+2^{q+1}\cdot\left(2^{s-1}-1\right) \\
\vdots & \vdots & \ddots & \vdots \\
b+2^{s}\cdot\left(2^{q-s+1}-1\right) & b+2^{s}\cdot\left(2^{q-s+1}-1\right)+2^{q+1} & \cdots & b+2^{s}\cdot\left(2^{q-s+1}-1\right)+2^{q+1}\cdot\left(2^{s-1}-1\right)
\end{array}$$

Figure 5: The accessed addresses with stride $S = 2^s$

$$\begin{array}{rr}
 & <a_{n-1}\ldots a_{q+s},\ a_{q+s-1}\ldots a_{s},\ a_{s-1}\ldots a_{q},\ a_{q-1}\ldots a_{0}> \\
+ & \delta_{q-1}\ldots\delta_{0} \\
\hline
 & <b_{n-1}\ldots b_{q+s},\ b_{q+s-1}\ldots b_{s},\ b_{s-1}\ldots b_{q},\ b_{q-1}\ldots b_{0}>
\end{array}$$

Figure 6: Binary bits representation of $b = a + \delta$ when $s > q$

c). Unit-stride access.

This has already been proved in Property 2.

III) $s > q$.

a) Strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

When $s > q$, as the SAMS scheme adopts the module assignment function from Harper's XOR scheme directly, therefore it is *strictly* conflict-free for strided access with stride family $\{S \| S = \sigma \cdot 2^s,\ \sigma\ odd\}$.

b) Unit-stride access.

To prove the SAMS scheme to be conflict-free for unit-stride access, we only have to prove that the conflicting items in linear address space are mapped to the same row. Depicted in mathematics, given two different addresses a and b, we need prove $r(a) = r(b)$ under the conditions $m(a) = m(b)$ and $1 \le |b - a| \le 2^q - 1$.

Assume $b = a + \delta(1 \le \delta \le 2^q - 1)$. Consider the binary bits representation of $b = a + \delta$, as depicted in Figure 6. From the figure we could see that, if $a_{q-1} \ldots a_0 = b_{q-1} \ldots b_0$, then $\delta = 0$, which means $a = b$. Therefore $a_{q-1} \ldots a_0 \ne b_{q-1} \ldots b_0$. In addition we know $m(a) = m(b)$, i.e.

$$\begin{cases}
a_0 \oplus a_s & = & b_0 \oplus b_s \\
a_1 \oplus a_{s+1} & = & b_1 \oplus b_{s+1} \\
\ldots & & \\
a_{q-1} \oplus a_{q+s-1} & = & b_{q-1} \oplus b_{q+s-1}
\end{cases}.$$

Therefore we know $a_{q+s-1} \ldots a_s \ne b_{q+s-1} \ldots b_s$. Consequently we know that there should be a carry input from bit $s - 1$ to bit s, which is the only possible way to make the equation in Figure 6 stand. Furthermore, as the carry outcome of bit $s - 1$ could only come from that of bit $q - 1$, therefore we have

$$\begin{aligned}
a_{s-1} \ldots a_q &= 1 \ldots 1 && (14) \\
b_{s-1} \ldots b_q &= 0 \ldots 0 && (15) \\
b_{n-1} \ldots b_q &= a_{n-1} \ldots a_q + 1\ . && (16)
\end{aligned}$$

As $b_q = 0$(from Equation (15)), we could further get

$$\frac{b_{n-1} \ldots b_q}{2} = \frac{b_{n-1} \ldots b_q + 1}{2}\ . \qquad (17)$$

Combining equation 16 and Equation 17, we know

$$\frac{a_{n-1} \ldots a_q + 1}{2} = \frac{b_{n-1} \ldots b_q + 1}{2}\ ,$$

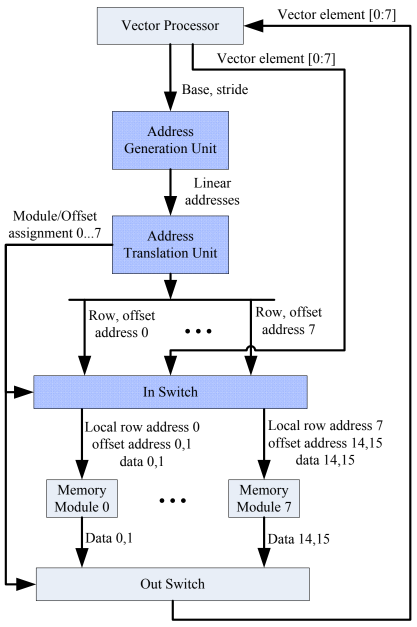


Figure 7: Parallel memory system based on the Matched SAMS Scheme

i.e.

$$\left(\frac{a}{2^q}+1\right)/2 \quad = \quad \left(\frac{b}{2^q}+1\right)/2 \ ,$$

which indicates that $r(a) = r(b)$(please refer to Section 3 for the definition of row assignment function). This means that any conflicting items(items located in the same memory module) under unit-stride access in the SAMS scheme are assigned to the same row, therefore they could be referenced simultaneously in one access. □

Corollary 1. The Matched SAMS Scheme is conflict-free for stride 1(unit stride), $2, \dots, 2^{q-1}$ and stride family $\{S \| S = \sigma \cdot 2^q, \ \sigma \ odd\}$.

Proof. In the Matched SAMS Scheme, the parameter s is set to q, therefore Corollary 1 is virtually the direct application of Property 2 and Theorem 2. □

The Matched SAMS Scheme is simple yet powerful, because of the large stride range it covers. In general, the Matched SAMS Scheme is capable of supporting conflict-free accesses with strides from $log_2(\#modules)+1$ families. For example, if we have a parallel memory system with 8 memory modules which deploys the Matched SAMS Scheme, then it could provide conflict-free access for unit stride, stride 2, stride 4, and any stride of $8 \cdot \sigma(\sigma \ odd)$. Thus the potential benefit is very promising in high performance vector processing systems, where a large number of processor clocks are spent on loading, packing and unpacking data from memory. With careful choice of design parameters and proper implementation trade-offs (we will discuss them in next section), the Matched SAMS Scheme could be utilized to accelerate the vector memory access in specific application domains such as scientific and engineering computing with high data-level parallelism.

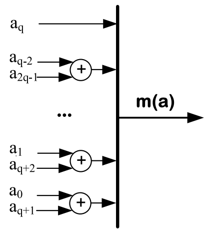


Figure 8: Module assignment function in Matched SAMS Scheme

5 Hardware Implementation and Discussions

Above, we have presented the mathematical foundations of the SAMS scheme, and particularly the Matched SAMS Scheme. For any parallel memory scheme to be practically useful, it is important to have efficient hardware implementation as the scheme logic is in the critical path of every memory access. In this section, we will examine the hardware implementation issues of the Matched SAMS Scheme. We generated the Verilog RTL source code of the SAMS memory subsystem with the help of our visual SAMS design tool and synthesized them by Synopsys Design Compiler tool chain (version A-2007.12) with SMTC 90 *nm* Low-K CMOS technology. Furthermore, we will also present some discussions about the variants of SAMS implementation with different address generation schemes and support for different write patterns, its overhead compared to the low order interleaving scheme, the scalability regarding the total memory capacity, its capability for unaligned memory access, and its potential applications. Unless explicitly stated, all implementations are done with parameters that the total capacity of all SAMS memory modules is 1MB, and data width of each vector element are 32 bits. Considering the hardware complexity and the popularity of vector strides, the supported access strides are $\{1, 2, 4, \ldots, 2^q\}$ (i.e. the strides $\sigma \cdot 2^q$ with $\sigma > 1$ are not supported in the implementations presented in this section).

Figure 7 illustrates the organization of parallel memory system based on our Matched SAMS Scheme. The vector processor core issues the memory access commands, with the base address and stride[6] to the Address Generation Unit(AGU), where the 2^q(in Figure 7 $2^q = 8$) linear addresses are calculated in parallel and then sent down to the SAMS memory system. The eight linear addresses are resolved by the Address Translation Unit(ATU) into eight module assignments, eight row addresses and eight row offset addresses. After that, the eight groups of row-offset pair and eight data items from input data port (on a memory write) go to the address & data switch and get routed to the proper memory modules according to their corresponding module assignments. In case of a read memory access, after the read latency of the memory modules[7] eight read data are fed back to the vector processor via the data switch at the bottom of Figure 7. It should be noted that in our current SAMS implementation, the delay starting from the moment when the vector memory access command is issued by the vector processor until the addresses and data reach the accessed memory modules, costs one cycle of the SAMS memory system (we will refer it as the "inbound path"), and the delay starting from the time when the read data are available at memory module ports until they arrive the vector processor takes another clock cycle (we will refer it as the "outbound path"). We have found that the critical path of the SAMS memory system is the inbound path, starting from the vector processor and ending at the SRAM memory modules. When we mention "SAMS memory system" or "SAMS implementation", we mean the entire memory system, which contains all the components excluding the vector processor in Figure 7. Also, when we say logic consumption, we mean logic consumption of entire SAMS memory system excluding the SRAM modules.

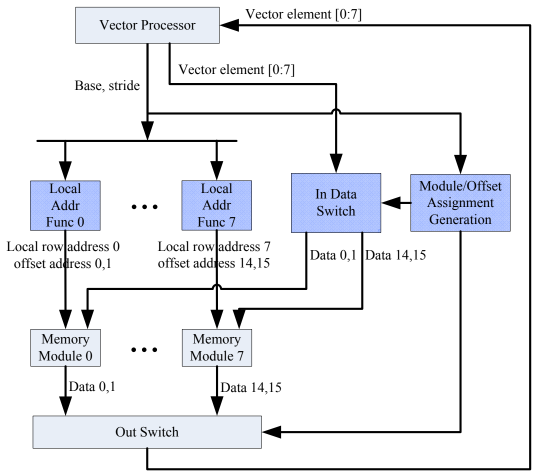


Figure 9: Matched SAMS Scheme with distributed address generation

5.1 Address Translation Unit

As shown in Figure 7, address translation plays a central role in a parallel memory scheme for two reasons. First, it determines the address mapping from the linear address seen by the vector processor to the physical locations; Secondly, it is constantly in the critial path of memory access. Thus we will first examine the address translation logic of the Matched SAMS Scheme. Checking the mathematical description of the address mapping procedure in Equation 1, it is clear that the row assignment function $m(a)$ and offset assignment function $o(a)$ are just static bits selections, which are fixed at system design time, therefore they don't involve any hardware logic. The hardware logic of the module assignment function is shown in Figure 8. We could see that the critical path of the module assignment function is a simple XOR gate, which is also the critical path of the address translation logic. The simplicity of the address translation logic is utmost desirable since it potentially allows for fast and efficient vector access at the system level.

5.2 Centralized Address Generation vs. Distributed Address Generation

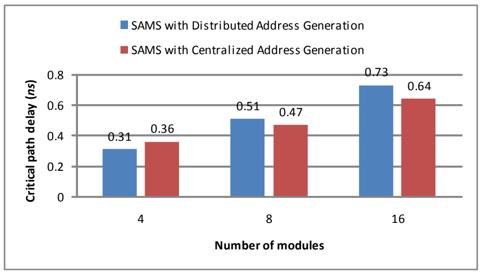
In general, there are two address generation schemes for vector processors: the centralized scheme and distributed scheme [15, 26, 25]. In the centralized scheme, the Address Generation Unit (AGU) generates all memory addresses to be accessed and then they are routed to the corresponding memory modules properly through an address routing circuit, as the In Switch shown in Figure 7[8]. In SAMS, the address routing circuit is a 2^q-to-2^{q+1} switch, which could be implemented with 2^{q+1} 2^q-to-1 multiplexors. In the distributed scheme, however, the vector memory access command (including information of base, stride, and vector length) is broadcasted to each and every memory module, and the row and offset addresses for all accessed memory modules are generated locally, in a distributed manner, as shown in Figure 9. The distributed address generation scheme normally has the advantage over its centralized counterpart that it does not require the address routing circuit, because local addresses are generated at the site of memory modules. We have implemented both address generation schemes for the Matched SAMS Scheme, and compared their performance in terms of critical path delay and logic consumption.

As we could see in Figure 10, the two address generation schemes did not show much difference in SAMS. The critical path delay of the distributed scheme is even slightly higher than that of the centralized

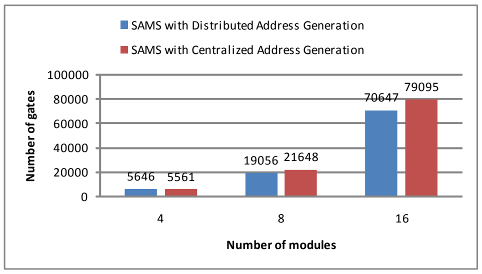
[6]The vector length here equals the number of memory modules.

[7]The latency of the memory module may be more than one clock cycle, depending on the absolute memory module latency (which is mainly determined by the memory capacity) and the critical path latency of the Matched SAMS Scheme. Here we assume the memory modules are fully pipelined.

[8]Note, the In Switch in Figure 7 routes not only addresses but also data from the vector processor.



(a)



(b)

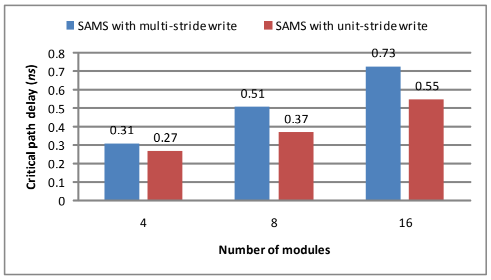
Figure 10: Centralized Address Generation vs. Distributed Address Generation

counterpart for implementations with 8 and 16 memory modules, although the area consumption is a little bit lower. One major reason why the distributed address generation didn't gain much benefit in SAMS implementations is that, although the address routing circuit is removed in the distributed address generation scheme, however, as we could see from Figure 7 and Figure 9, the input data switch circuit, which routes data from the vector processor to memory modules, are kept untouched in both SAMS implementations. Actually the critical path in the SAMS implementation with distributed address generation is exactly the data routing circuit, which consists of the In Data Switch unit and the routing information generator, i.e. the Module/Offset Assignment Generation unit. Besides the critical path delay, the data routing network also contributes to the majority of the overall logic consumption. In fact, in almost all SAMS implementations, around 80% of the logic consumption exclusively comes from the input and output data switches.

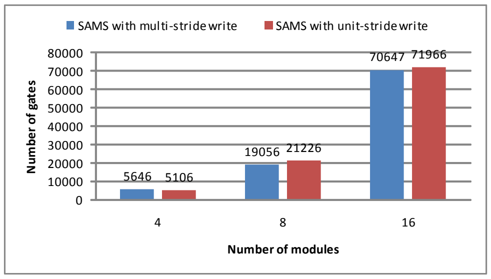
5.3 Unit-Stride Memory Write vs Multi-Stride Memory Write

The above implementation supports both unit-stride and strided vector stores. Through our initial investigation into real applications, we found that the multi-stride memory write pattern is not frequently used. If the set of supported strided memory write patterns is reduced, it is hopeful that the complexity of input data routing circuit could also be reduced. This could be quite beneficial for the entire SAMS memory system, as the input data switch consumes the largest portion of the overall logic consumption, and more importantly, its latency also dominates the inbound path, which is the critical path of the SAMS implementations. Therefore another trade-off in the implementation of the Matched SAMS Scheme could be such as to reduce the hardware complexity by supporting only the unit-stride memory write pattern. Using the distributed address generation scheme, the critical path delay and logic consumption of the two SAMS implementations with unit-stride memory write support and multi-stride memory write support are compared in Figure 11.

We could see from the figure that the critical path delay is decreased when SAMS is implemented to support only unit-stride vector stores. The reduction in critical path delay is trivial for the case of SAMS implementation with 4 memory modules; However, it is quite significant for the cases with 8 modules (around 27%) and 16 modules (around 25%). The difference in logic consumption is negligible. Although Figure 11 only shows the difference between SAMS implementations with unit stride write support only and support for all strides write in $\{1, 2, 4, \ldots, 2^q\}$, it is also possible to choose to support a subset of the entire stride spectrum. Consequently, different design options are available at different points in the



(a)



(b)

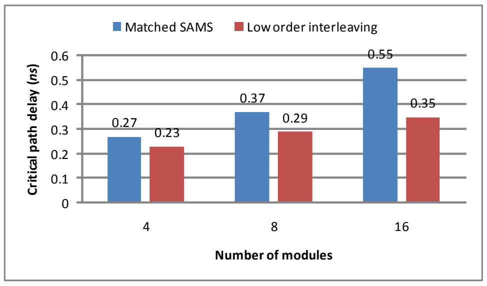
Figure 11: Unit-stride memory write vs. multi-stride memory write

supported write patterns - hardware complexity curve, and choices could be made for vector processors targeting different application domains. Similarly, support for strided read patterns could also be tailored to meet the performance - cost specifications of different applications.

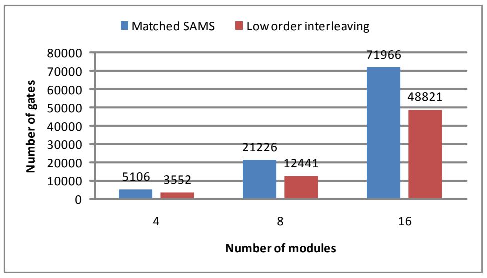
5.4 Overhead Compared with Low Order Interleaving

With many parallel memory schemes proposed in literature, the most popular one is still the simple low order interleaving scheme in engineering practice [32, 17, 21], which maps data elements with consecutive linear addresses to consecutive memory modules. As already shown in Figure 1, the low order interleaving scheme is unable to support conflict-free even strided vector access. The Matched SAMS Scheme comes up with the capability of providing conflict-free memory access for more access patterns, inevitably at certain hardware cost. To see the hardware cost of the extra functionality more clearly, we have implemented low order interleaving scheme with the same systematic parameters as those in the Matched SAMS Scheme with distributed address generation and unit-stride write support. Figure 12 shows the critical path delay and logic consumption for both Matched SAMS and low order interleaving schemes.

We could see from Figure 12 that there is some moderate increase in the critical path delay, and a substantial increase in the logic consumption. We have also identified that the increases in both sides come from the input and output data switch. In fact, if the implementation of a low order interleaving scheme needs a $2^q \times 2^q$ crossbar, then a $2^{q+1} \times 2^q$ crossbar is required in the Matched SAMS Scheme. Therefore, the penalty of the Matched SAMS Scheme is much the same as the use of redundant memory modules, as far as the data alignment logic is concerned. However, as the Matched SAMS Scheme uses less memory modules than redundant memory schemes, memory module resources incurred by extra memory modules such as the address wires and decoding logic could be saved, thus the Matched SAMS Scheme potentially allows for higher density of memory arrays, which preferable in CMOS process technologies.

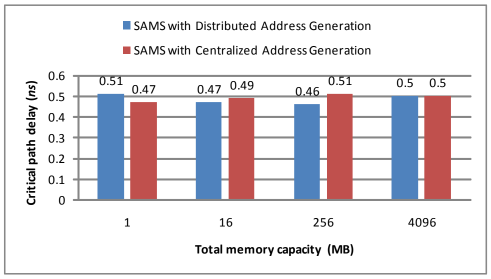


(a)

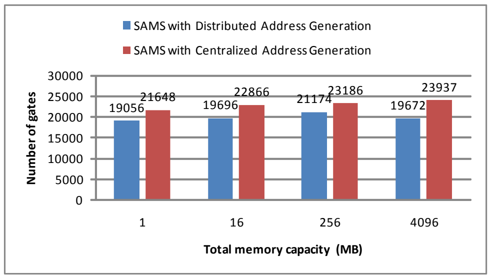


(b)

Figure 12: Overhead compared with low order interleaving



(a)



(b)

Figure 13: Scalability with regard to memory capacity

5.5 Scalability with Regard to Memory Capacity

Above we assume that the total capacity of SAMS memory system is 1MB. Now we will examine the scalability of different SAMS implementations, as shown in Figure 13. Here we set the number of memory modules to 8, and both SAMS implementations with distribute and centralized address generation support multiple stride vector stores. As we could see from Figure 13, the fluctuation in the critical path delay and logic consumption in both SAMS implementations is almost negligible when the entire memory capacity is increased from 1MB to 4GB. Actually when we look back into Equation 1, we could find out that none of the logic of the module assignment function, row address function and offset assignment function is related to n (the number of bits of the SAMS memory system address space), which means that the ATU in Figure 7 is not related to the memory capacity. It is also clear that the input and output switches are not relevant to the memory capacity. Therefore, only the AGU is related to n, where the stride (or multiple of stride) is added to the base address to generate the addresses for vector elements, and the increases in the critical path delay and logic consumption come from the adders. Note, as we choose to support only the strides $\{1, 2, 4, \ldots, 2^q\}$, the adders are not full n-bit adder thus the logic could be quite simplified. In fact, the AGU accounts for a very small portion of the entire logic consumption (much less than 10%). Therefore the increase in n doesn't remarkably impact the critical path delay and the logic consumption of the entire SAMS memory system. It should be noted also that, although there is no AGU in Figure 9, the addition logic to generate the proper vector element addresses is scattered into local address functions (Local Addr Func $0, \ldots, 7$) and the Module/Offset Assignment Generation logic.

5.6 Unaligned Vector Access

Unaligned vector memory access is one of the critical problems in SIMD processing systems [27, 31]. It should be noted that all SAMS implementations described above, are capable of supporting unaligned vector access. The SAMS implementation with multi-stride memory write capability supports unaligned unit-stride and strided vector load and stores; Whereas SAMS with unit-stride memory write capability supports unaligned unit-stride and strided vector loads and unaligned unit-stride stores. Additionally, with multiple independent memory modules as backup storage, the store granularity could be reduced from 2^q elements to a single one, compared to the single memory module implementation. For example, the monolithic local store of IBM Cell SPE only supports loads/stores at the granularity of 128 bits (that could be, 4 integers/floating point numbers or 2 double precision numbers); while with the Matched SAMS Scheme with 4 memory modules, the store granularity could be reduced to 1 integer/floating point number, and stores of 1/2/4 32 bits data (that could be, 1/2/4 integers/floating point numbers or 1/2 double precision numbers) are all supported.

5.7 Applications

Regarding the application, the Matched SAMS Scheme could be applicable wherever the data level parallelism is exploited in the form of (regular) vector processing, to boost the performance of vector processing with multiple stride memory accesses. First, it could be considered in memory subsystem in traditional vector supercomputers, where the system performance could benefit from the high memory throughput provided by SAMS. Moreover, it is quite common that there are many on-chip memory modules even in mainstream microprocessors, as on-chip SRAM arrays are usually split into a group of subarrays, in order to reduce the memory access latency or to pipeline the memory access [30, 12]. Therefore, the Matched SAMS memory system could also be adopted as on-chip storage in microprocessors. For example, it could be used in the on-chip data memory for SIMD processors, such as SPU Local Stores (LS) in the Cell processor [19], to improve the flexibility of vector memory accesses. It could also be considered for integration as on-chip data buffer for GPP SIMD extensions, where the data alignment and permutation problems, which result from the lack of flexible memory access support, remain to be the major bottlenecks for many applications [27, 31]. It should be noted that the integration of a SIMD buffer into a GPP introduces coherence problem, as it will be deployed at the same level as the data caches in memory hierarchy. However, this problem could be solved by either snooping mechanisms or by the use of non-cacheable regions in the address space.

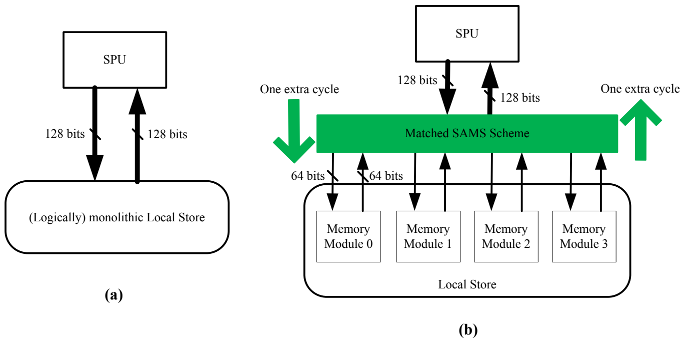


Figure 14: Integration of Matched SAMS Scheme into Cell SPE

Table 1: Selected benchmark suit

Program	**Source**	**Type**	**Application Domain**	**Working Set**
streamcluster	PARSEC	application	data mining	medium
fluidanimate	PARSEC	kernel	animation	large
complex multiplication	Cell SDK	micro-kernel	-	small
matrix transpose	Cell SDK	micro-kernel	-	small

6 Experimental Evaluation

The manipulation of data movement between the processing elements and the storage, including data loads/stores and permutations, has long been a major challenge in high-performance data-parallel processing systems design. For example, data permutation and alignment problems are among the critical bottlenecks which undermine the performance of GPP SIMD extensions [27, 31]. The Matched SAMS Scheme proposed in this paper provides efficient hardware support for unaligned multi-strided memory accesses with improved store granularity, therefore it has the potential of alleviating the data manipulation penalties and consequently freeing the power of SIMD processing.

To validate the performance of proposed Matched SAMS parallel memory scheme in real applications, we applied it to the IBM Cell processor, which is designed for computation-intensive applications with high data-level parallelism [19, 29]. The integration of our parallel memory scheme into Cell processor is illustrated in Figure 14. Figure 14(a) shows the original local store memory organization in Cell SPE; while Figure 14(b) shows the modified local store memory hierarchy with the integration of the Matched SAMS Scheme. Note only the data buses for local store accesses are shown in Figure 14, and the 1024-bit data port between LS and the DMA engine in SPE [14] remains untouched therefore it is not shown there. It should also be noted that the total size of the four memory modules in Figure 14(b) is kept the same as the original local store size.

6.1 Experimental Setup

We use CellSim developed at BSC [2], which is a cycle-level full system simulator for IBM Cell/BE processor. The benchmarks of our experiments consist of some full applications from PARSEC [1], and some micro-kernels from IBM Cell SDK [4] as well. Table 1 lists the major features of the selected benchmarks. Streamcluster from PARSEC is an online clustering kernel which takes streaming data points as input and uses a modified k-median algorithm to do the online clustering. The parameters of streamcluster workload in our study are set as follows: 2048 input points, block size 512 points, 10 point dimensions, 5-10 centers, up to 500 intermediate centers allowed. Fluidanimate is an Intel RMS application and it uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes [7]. The fluidanimate workload in our experiments uses the *simsmall* input set provided

Table 2: Changes over original SPE and their impact on overall performance

+/-	**Changes over original SPE**	**Impact on overall performance**
-	Latency of store is increased from 4 clock cycles to 5 cycles	Increase in store latency could be negligible in SPU context with the help of large register file and a modern compiler such as gcc
-	Latency of load is increased from 6 cycles to 8 cycles	Increase in load latency (33%) could significantly degrade system performance if instruction scheduling could not hide the load latency
-	Penalty of a taken branch is increased from 18 cycles to 20 cycles	Increase in branch penalty is around 11%. Considering the fact that the portion of (taken) branches in the total amount of executed instructions is not likely to be very large in applications targeted by Cell, this impact on overall performance might be moderate
-	Hardware overhead	The logic consumption of the SAMS scheme implemented in the new SPE equals approximately to that of 3 32-bit adders/subtracters[9]
+	Reduction of # of executed load/store instructions as a result of direct support for unaligned and strided memory access	Reduction in execution time and memory traffic. (This reduction is particularly favorable for Cell SPE since reduction in LS traffic also decreases the chance of LS port conflict (remember LS is single port memory which is shared by data load/store, instruction fetch and DMA transfer))
+	Reduction of "glue" instructions which are indispensable for realignment and pack/unpack tasks	The reduction in execution time may be considerable, since the # of glue instructions are usually nontrivial.
+	Harware (and architectural and ISA level) support for unaligned and strided memory access	Programmer & compiler -friendly: since HW takes care of much of the data alignment and pack/unpack job , it relieves programmers and compilers from such burden and leaves more opportunities of higher-level optimizations for them

by PARSEC. The 3D working domain has been shrunk and the maximal number of particles inside a cell has been reduced to 6, to fit the relatively small local store size of SPE in our study. The simulation runs for one timestep to compute one frame. Streamcluster and fluidanimate are chosen for our experiments because they are applications with high level data parallelism at fine- to medium- granularities (as reported in Table 1 in [7]), which fits well to the target application domains of the Cell processor. Besides full applications, we also include some micro kernels, including complex number multiplication and 4x4 matrix transpose. We take the source code of complex number multiplication from [3]. The workload of the complex multiplication experiment is set to 10K multiplications. Matrix transpose is frequently used in matrix algebra. The 4x4 matrix transpose is particularly important for 4-way SIMD paradigms, for example, it is used in 2D-FFT kernel in Cell SDK. We took the source code of 4x4 matrix transpose from IBM Cell SDK library [4]. The work load of the matrix transpose is set to 10K transposes. To compile the C code, we use a compiler suit comprising two stand-alone compilers: ppu32-gcc for PPU (which is from PPU toolchain 2.3 based on gcc version 3.4.1) and spu-gcc for SPU (which is from SPU toolchain 3.3 based on gcc version 4.1.1). The two micro-kernels are compiled with -O1 optimization option.

One of the major impact to the SPU microarchitecture with the incorporation of the Matched SAMS Scheme into the SPU local store memory hierarchy is that, the SPU load/store pipeline is lengthened, since the parallel memory scheme introduces additional delay for scheme logic. In our experiments, we choose to implement the distributed address generation and unit-stride memory write schemes, as discussed in Section 5.2. According to our synthesis results in Table **??**, the logic delay of the Matched SAMS Scheme implementation with distributed address generation and unit-stride memory write for 4 memory modules is 0.27 ns. With the same technology and synthesis tool, the critical path of a 32-bit signed adder/subtracter is 0.52 ns, which indicates that the critical path delay of the Matched SAMS Scheme is around half of that of the 32-bit signed adder/subtracter in our study. In Cell SPU, the fixed-point ALU is pipelined into two stages, therefore we project the deployment of the Matched SAMS Scheme in SPE's local store will introduce one additional pipeline stage for the inbound path, and another pipeline stage for the outbound path. As the load and store instructions take 6 and 4 clock cycles to accomplish respectively [29], they will cost 8 and 5 clock cycles in the new pipeline with integration of our parallel memory scheme.

Although, there are major difficulties in projecting the accurate circuit delay of the Matched SAMS

Table 3: New instructions and intrinsics added to CellSim and spu-gcc

Name	**Type**	**Operation**	**Alignment (Bytes)**
lqd ‡	instruction	load a quad word (QW) with unit stride, d-form	4
lqx ‡	instruction	load a quad word (QW) with unit stride, x-form	4
lqa ‡	instruction	load a quad word (QW) with unit stride, a-form	4
lqr ‡	instruction	load a quad word (QW) with unit stride, r-form	4
lqds2	instruction	load a quad word (QW) with stride 2, d-form	4
lqxs2	instruction	load a quad word (QW) with stride 2, x-form	4
lqas2	instruction	load a quad word (QW) with stride 2, a-form	4
lqrs2	instruction	load a quad word (QW) with stride 2, r-form	4
lqds4	instruction	load a quad word (QW) with stride 4, d-form	4
lqxs4	instruction	load a quad word (QW) with stride 4, x-form	4
lqas4	instruction	load a quad word (QW) with stride 4, a-form	4
lqrs4	instruction	load a quad word (QW) with stride 4, r-form	4
stwd	instruction	store a word, d-form	4
stwx	instruction	store a word, x-form	4
stwa	instruction	store a word, a-form	4
stwr	instruction	store a word, r-form	4
stdwd	instruction	store a double word (DW), d-form	8
stdwx	instruction	store a double word (DW), x-form	8
stdwa	instruction	store a double word (DW), a-form	8
stdwr	instruction	store a double word (DW), r-form	8
stqd ‡	instruction	store a quad word (QW), d-form	4
stqx ‡	instruction	store a quad word (QW), x-form	4
stqa ‡	instruction	store a quad word (QW), a-form	4
stqr ‡	instruction	store a quad word (QW), r-form	4
spu_lqs2(ra, rb)	C intrinsic	load a QW at base address (ra+rb) with stride 2; unified intrinsic for lqds2, lqxs2 and lqas2	4
spu_lqs4(ra, rb)	C intrinsic	load a QW at base address (ra+rb) with stride 4; unified intrinsic for lqds4, lqxs4 and lqas4	4

‡: These instructions are taken from the original SPU ISA, with the alignment changed from 16 bytes to 4 bytes with the integration of the Matched SAMS Scheme.

Scheme logic on the Cell processor die, such as the significant differences between synthesis results using standard cell library and full-custom circuit implementation, and the different characteristics of the Matched SAMS Scheme logic (which is dominated by the crossbar circuitry, as discussed in Section 5) and the adder logic (which is fundamentally a parallel-prefix problem [37]), nevertheless we use the adder-based projection described above because of its simplicity. It should be noted that the local store memory array is already split into submodules and accesses to them are fully pipelined in the SPE implementation [12], therefore the implementation of four independent memory modules based on current Cell physical design may not be a problem; furthermore, there is a potential of implementing the memory scheme logic in parallel with the original partial address decoding logic and data address routing wires, to hide the Matched SAMS Scheme logic delay inside the long memory access pipeline. Taking these factors into account, we feel that the above 8 cycles load and 5 cycles store may be some realistic estimation for the extra delays incurred by the integration of our proposed memory scheme inside the SPU pipeline.

To make the enhanced load/store capabilities resulted from our Matched SAMS Scheme available to software, we have extended the SPU ISA by introducing a set of new instructions for strided and unaligned memory accesses. We have also created the required C intrinsics that facilitate the use of strided memory access instructions in C programming. Table 3 shows all new instructions and C intrinsics. To reflect the changes in architecture, we have modified the spu-gcc backend to make it capable of analyzing the unaligned memory access requirement and generating proper code with the newly added instructions. The load latency has also been updated to 8 in spu-gcc for proper code scheduling with the pipeline change. Besides the compiler, the CellSim has also been modified to adopt all the changes.

Before digging into the experiments, we now list all major changes of our new[10] SPE model over the original SPE and their impact on overall performance in Table 2. In the table "+" denotes positive effects and "-" means negative effects. From the preliminary analysis we could get the impression that the Matched SAMS Scheme could be beneficial for applications with data alignment problem in Cell SPE.

It should be noted that although there are eight SPEs available in Cell processor, we only use a single one in our experiments. This is because we want to focus on the performance impact of the Matched SAMS Scheme in SIMD processing elements, instead of exploiting the extent to which our benchmarks could be parallelized on the eight SPEs. Virtually, the advantages introduced by our parallel memory scheme within a single SPE can be extended to all SPEs in Cell.

6.2 Parallelization of Benchmarks

The PARSEC benchmarks are written using C++ and pthread library, with the aim of measuring the system-level performance of multi-core processors. Since the aim of our experiments is to reveal the performance of the Matched SAMS Scheme on SIMD devices like the Cell SPU, we use the single thread version of the source code and run the applications with a single SPU. In both applications of streamcluster and fluidanimate, the data preparing task and OS interface (such as file IO) is implemented in PPU, while majority of the data processing is offloaded to SPU. After data has been ready in main storage, the PPU triggers SPU to start processing. Since large arrays are allocated in Cell's main memory, SPU reads a portion of them each time via DMA transfers, process them and then writes the temporary results back to the main storage again by DMA transfers (if the temporary results are too large to be held in LS). It continues by reading the next chunk of data and so on.

The greatest modifications to the source code come with hand SIMDization of the code inside the critical loops. For SIMDization, we use SPU C instrinsics instead of assembly code as we want to designate specific SPU instructions while leaving low-level optimizations such as register allocation and instruction scheduling to the compiler. Original algorithms and major data layout in the source code have been respected, with exceptions that necessary changes in data layout and accordingly the dataflow have been made in order to fit the relatively small LS size of the SPE.

[10]Hereafter we will refer to the SPE model with the integration of Matched SAMS Scheme in its local store as the *new* SPE, and the baseline standard Cell SPE as the *original* one.

Table 4: Comparison of SPU dynamic instruction count and execution time

Name	**Memory Instruction Count**			**Total Instruction Count**			**Execution Time (cycles)**		
	Original (load/store)	New (load/store)	↓	Original	New	↓	Original	New	↓
streamcluster	319,188,572 (231,429,310/ 87,759,262)	174,355,761 (132,889,953/ 41,465,808)	45%	754,225,152	410,966,630	46%	3,118,623,559	2,056,226,698	34%
fluidanimate	95,528,046 (67,734,900/ 27,793,146)	67,786,359 (48,001,912/ 19,784,447)	29%	257,433,162	237,021,420	8%	1,052,323,740	1,028,511,341	2%
complex multiplica-tion	15,392 (10,256/5,136)	15,421 (10,274/5,147)	0%	42,008	31,815	24%	96,242	84,602	12%
matrix transpose	81,950 (40,974/40,976)	81,979 (40,992/40,987)	0%	167,766	85,893	49%	283,121	154,042	46%

For the micro-kernels of complex number multiplication and 4x4 matrix transpose, to reduce the impact of DMA transfer overhead on the performance comparison between the baseline and enhanced SPE, we assume an in-place computing paradigm, in which the kernel functions processes the data at the same LS address. In this way DMA transfers are avoided in our experiments for micro-kernels.

It should be noted that, for all experiments, the source C codes for both original SPE and new SPE versions are kept the same. The only exceptions are: 1)glue code for handling the unaligned memory access in original SPE are removed from new SPE since they are no longer necessary with the new architecture, and 2) the C instrinsics in Table 3 are explicitly used in new SPE context where there are performance gains by using them. The specific instructions with the new SPE for unaligned memory access and small granularity memory access (such as store a word or double word) are invoked automatically by our modified spu-gcc compiler when appropriate.

6.3 Experimental Results

We measure the performance by counting the application execution time on SPU. Note the entire execution time of the application also includes the portion of PPU execution time which is not overlapped with SPU. Nevertheless, we only compare the SPU execution time since the PPU time is small and it is the same for both original and new SPE configurations in our experiments. Table 4 shows the performance comparison of the original SPE and the new one with our Matched SAMS parallel memory scheme integrated in its local store memory hierarchy. We could see that the major performance gain comes from the reduction of executed instruction count (either load/store instructions or glue instructions, or both).

The most critical loop in streamcluster source code is to calculate the distance between current point and one of the candidate cluster centers. Therefore, loading the point data (both for current point and the candidate center) efficiently is the key to good performance for SIMD paradigm. Unfortunately, the point data are not normally aligned to 16 byte boundary (alignment is guaranteed only when the dimension is multiple of 4). As a consequence, with the original SPE we have to explicitly realign the point data, which incurs not only more loads and stores but also more glue instructions compared with the new SPE version, as shown in Table 4. To summarize, the new SPE gains dramatic benefits from the capability of accessing the unaligned point data directly supported by our Matched SAMS Scheme.

In fluidanimate, most of the execution time is used to compute the density and the consequent force between two particles within effective range in two neighbor cells, which is inside a 6-level nested loop with 3 conditional jumps. Each of the particle data comprise of fields such as position, density, velocity, acceleration (which furthermore consists of 3D or 1D float numbers), and they are not aligned to 16B memory addresses. Again, the alignment problem occurs here. Similar to streamcluster, the amount of memory accesses is remarkably reduced by the unaligned vector access capability in fluidanimate. However, since the critical loop consists of 6 nested loops and 3 conditional jumps, the weakness of SPU with branchy code is magnified here, which leads to a huge number of cycles wasted on pipeline stall due to taken branches. As a consequence, the contribution from the reduction in memory instructions to the reduction in total executed instruction count is amortized by frequent pipeline stalls. The reduction in total execution time is further undermined by the increase in branch penalty with new SPE (as described in Table 2).

In complex number multiplication, with the capability of unaligned stride-2 vector access, the real (in-phase) vector and imaginary (quadric) vector could be loaded directly, instead of loading the mixture of them and extracting the real and imaginary parts using a sequence of shuffle instructions, as the source code in [3] did. Therefore, although the memory access count is the same for both original and new SPEs, the kernel still achieved some performance gains with new SPE as a result of the reduction of the glue instructions.

The benefits for 4x4 matrix transpose from our SAMS scheme is quite clear: with stride-4 vector access capability, the transpose procedure is simple accomplished in a straightforward way: first load 4 columns of the input matrix directly into register and then store them to the rows of output buffer. Intermediate shuffle processing in original code is completely eliminated. This explains the huge performance gain with new SPE in our experiments.

In conclusion, experiment results demonstrate that adoption of the Matched SAMS Scheme is a feasible solution for data alignment problem in SIMD processors. We feel that our parallel memory scheme brings SIMD devices such as Cell SPE with two major advantages: 1) it eases the programming/compilation procedure of the SIMD systems (this is what we feel during the parallelization of the benchmarks), and 2) it provides substantial overall performance improvement for applications with bottlenecks in unaligned and strided memory access. It should also be noted that, although there are programming techniques such as restructuring the data organization (e.g. using structure of arrays (SOA) instead of array of structures (AOS)) to relieve the data alignment problem. However, we feel that our parallel memory scheme is still useful not only because it provides a hardware solution for the problem, but also because in some cases, such data layout restructuring is either not helpful or too expensive. For example, in fluidanimate the neighbor cells are determined according to current cell's 3D location, and they are not contiguous in 1D linear address space. Moreover, the number of particles in each neighbor cell is different. Consequently, accesses to particles in cells are not contiguous and access granularity is on single particle basis. In such occasions, reorganization of particle data structure could not solve the alignment problem.

7 Related Work

To cope with module conflicts of vector accesses across stride families, several techniques have been proposed in the literature, including the use of buffers [9], dynamic memory schemes [11, 10], memory modules clustering [9] and intra-stream out-of-order access [36].

The use of buffers [9] is probably most straightforward solution as it tolerates the module conflicts by simply buffering the input addresses and output data and collecting the required data after some delay. The buffer depth required depends on misalignment between the parallel memory scheme used and the vector access stride. If the access stream is distributed evenly between the memory modules of the system, then the peak throughput of one data per memory module in one cycle might be achieved after a transient startup time. However, since the startup disparity is unavoidable, this solution introduces significant time penalties in case of short access streams. Moreover, the use of buffers and the logic for collecting the correct data items from the buffers could cause substantial hardware overheads.

The dynamic scheme proposed in [11, 10] works well only when the same data set is accessed with single stride family. However, if the data set is to be accessed using different stride families, the penalty of flushing and reloading data between the memory modules and the lower level in the memory hierarchy may not be amortized in some cases, which would result in performance degradation of the system.

The memory modules clustering [9] introduces inefficient use of module control logic and data routing resources, as a large portion of memory modules may remain idle during each parallel memory access. For instance, under the assumption that the number of modules is a power of two number, the amount of memory modules used for conflict-free access of two unmatched stride families may be no more than 50% of the available modules. This results in waste of logic resources and power in some cases.

The out-of-order vector access [36] is based on the observation that a long, strided memory reference stream with module conflicts in sequential order could become conflict-free, if properly reordered. For instance, in a multimodule system with conflict-free stride-4 access support, a stride-2 stream with 16 memory references could be accomplished by two stride-4 streams with 8 memory references each. Basically the original stride-2 stream is split into two stride-4 sub-streams and the memory system is accessed with by the alternating sub-streams. In this case, the access is conflict-free. The problem with intra-vector out-of-order access is that it requires long vectors for proper operation. In addition, as data items are read out of order, data permutation logic may introduce additional penalties[11].

The most distinctive aspect of our SAMS scheme compared to the previous solutions for strided vector access is that it avoids the module conflicts when the memory reference patterns go back and forth between unit-stride and strided accesses, and thus truly-parallel data access is supported. Unlike the out-of-order vector access scheme, our proposal preserves the data sequence required by the vector load/store units, thus atomic parallel access is achieved for short vectors and peak performance could be sustained for vectors as short as 2^q elements. The SAMS proposal is a memory scheme with no module redundancy and high utilization of module resources. On the other hand, the SAMS Scheme is complementary to the existing techniques, which means that it could also take their advantages to improve system performance.

Regarding the data alignment problem in GPP SIMD extensions, studies have been done to boost the performance of SIMD devices by relieving the impact of non-contiguous and unaligned memory access from both the compiler and hardware (architecture) point of view. For example, Ren et al proposed a compiler framework to optimize data permutation operations by reducing the number of permutations in the source code with techniques such as permutation propagation and reduction [31]. Nuzman et al developed an auto-vectorization compilation scheme for interleaved data access with constant strides that are power of 2 [27]. Alvarez et al analyzed the performance impact of extending the Altivec SIMD ISA with unaligned memory access support on H.264/AVC codec applications [5]. Although they employed a simple memory scheme with two banks in L1 data cache to give support to unaligned loads/stores, they didn't consider strided memory access in their scheme.

SAMS was proposed to simultaneously support conflict-free unit-stride and strided memory accesses from one *single* stride family in [16], by first constructing a single-affiliation interleaving scheme, and then making data lines wider to solve the module conflicting problem in unit-stride access. In this paper, we propose the Matched SAMS Scheme to give support to conflict-free vector access for strides from $log_2(\#modules) + 1$ stride families, compared to 2 in [16]. We have also implemented the entire memory system based on the Matched SAMS Scheme in this paper. The synthesis results of the hardware implementation are given, and the integration of our Matched SAMS Scheme to IBM Cell SPE is proposed based on the synthesis results. Performance of the scheme with kernels and real applications is also investigated in this paper.

8 Conclusion

In this paper, we propose the Matched SAMS Scheme, which improves the previous memory interleaving schemes by providing conflict-free access for multiple stride families. The Matched SAMS Scheme is based upon SAMS; However it goes a step further by supporting conflict-free accesses with strides from $log_2(\#modules) + 1$ stride families, compared to 2 in the original SAMS scheme. We have presented the mathematical foundations for both schemes. We have also explored a variety of Matched SAMS implementa-

[11]Data permutation is not required in the original proposal [36] as there the assumed memory organization is that single datum is read out from the multiple memory modules per cycle, whereas in the organization considered in this paper multiple data items (equal to the number of memory modules) are read per cycle.

tions including the centralized and distributed address generation schemes, SAMS with unit-stride write and multi-stride write support, and compared their impact on the critical path delay and area consumption based on hardware synthesis results. We also compared the performance of SAMS implementation with distributed address generation and unit-stride write with that of the low order interleaving scheme, which is most popular in engineering practice. Based on hardware implementation results, we proposed to apply our Matched SAMS Scheme to IBM Cell processor by integrating it into the Cell SPE local store, and investigated its performance with some kernels and real applications using simulation. Our experimental results indicate that the Matched SAMS memory system has efficient hardware implementation and adequate performance for real applications, which makes it a promising technique for data-parallel processing systems.

9 Acknowledgments

This research has been supported by the European Commission in the context of the FP6 project SARC (FET-27648) and by the Dutch Technology Foundation STW, applied science division of NWO and the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533).

References

[1] http://parsec.cs.princeton.edu/index.htm.

[2] http://pcsostres.ac.upc.edu/cellsim/doku.php.

[3] http://www.cc.gatech.edu/ bader/cell/day1-06developingcodeforcell-simd.ppt.

[4] http://www.ibm.com/developerworks/power/cell/.

[5] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. Performance impact of unaligned memory operations in simd extensions for video codec applications. *ispass*, 0:62–71, 2007.

[6] D. Baily. Vector computer memory bank contention. *IEEE Trans. Computers*, 36:293–298, Mar. 1987.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. *Technical Reports*, TR-811-08:22, January 2008.

[8] P. Budnik and D. Kuck. The organization and use of parallel memories. *IEEE Trans. Computers*, C(20):1566–1569, Dec. 1971.

[9] D. T. Harper III. Block, multistride vector and FFT accesses in parallel memory systems. *IEEE Trans. Parallel and Distributed Systems*, 2(1):43–51, 1991.

[10] D. T. Harper III. Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Trans. Computers*, 41(2):227–230, 1992.

[11] D. T. Harper III and D. A. Linebarger. Conflict-free vector access using a dynamic storage scheme. *IEEE Trans. Computers*, 40(3):276–283, 1991.

[12] S. e. a. Dhong. A 4.8GHz fully pipelined embedded SRAM in the streaming processor of a CELL processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005*, pages 486–612, 2005.

[13] R. Espasa, M. Valero, and J. E. Smith. Vector architectures: past, present and future. In *Proceedings of the 12th international conference on Supercomputing*, pages 425–432, 1987.

[14] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit fors a cell processor. In *Proceedings of IEEE Int'l Solid-State Circuits Conference 2005*, pages 134–135, 2005.

[15] C. Galuzzi, C. Gou, D. Caldern, G. N. Gaydadjiev, and S. Vassiliadis. High-bandwidth address generation unit. *accepted for publication in Journal of VLSI*, December 2008.

[16] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev. SAMS: Single-affiliation multiple-stride parallel memory scheme. In *Proceeding of Computing Frontiers*, May 2008.

[17] S. Hammond, R. Loft, and P. Tannenbaum. Architecture and application: The performance of the NEC SX-4 on the NCAR benchmark suite. In *Proceedings of the ACM/IEEE Conference On Supercomputing 1996*, pages 22–22, 1996.

[18] W.-C. Hsu and J. Smith. Performance of cached DRAM organizations in vector supercomputers. *ACM SIGARCH Computer Architecture News*, 21:327–336, May 1993.

[19] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. & Dev.*, 49(4/5):589–604, 2005.

[20] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, pages 66–76, Mar. 2003.

[21] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, pages 21–29, Mar. 2005.

[22] L. Kontothanassis, R. Sugumar, G. Faanes, J. Smith, and M. Scott. Cache performance in vector supercomputers. In *Proceedings of the ACM/IEEE Conference On Supercomputing 1994*, pages 255–264, 1994.

[23] C. Kozyrakis. *Scalable Vector Media-Processors for Embedded Systems.* PhD thesis, UC Berkeley, Berkeley, CA, USA, May 2002.

[24] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER 6 microarchitecture. *IBM J. Res. & Dev.*, 51(6):639–662, 2007.

[25] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis. Algorithmic foundations for a parallel vector access memory system. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 156–165, New York, NY, USA, 2000. ACM.

[26] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis. Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 39–48, Jan 2000.

[27] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 132–143, New York, NY, USA, 2006. ACM.

[28] A. Pajuelo, A. Gonzalez, and M. Valero. Speculative dynamic vectorization. In *Proceedings of the 29th Ann. Int'l Symp. Computer Architecture*, pages 271–280, 2002.

[29] D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, H. Harvey, P.M. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41:179–196, 2005.

[30] D. W. Plass and Y. H. Chan. IBM POWER 6 SRAM arrays. *IBM J. Res. & Dev.*, 51(6):747–756, 2007.

[31] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. *SIGPLAN Not.*, 41(6):118–131, 2006.

[32] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, pages 63–72, Jan. 1978.

[33] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER 5 system microarchitecture. *IBM J. Res. & Dev.*, 49(4/5):505–521, 2005.

[34] T. Sun and Q. Yang. A comparative analysis of cache designs for vector processing. *IEEE Trans. Computers*, 48:331–344, Mar. 1999.

[35] J. M. Tendler, J. S. Dodson, J. S. Fields, H. L. Jr., and B. Sinharoy. POWER 4 system microarchitecture. *IBM J. Res. & Dev.*, 46(1):5–25, Jan. 2002.

[36] M. Valero, T. Lang, M. Peiron, and E. Ayguade. Conflict-free access for streams in multimodule memories. *IEEE Trans. Computers*, 44:634–646, 1995.

[37] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Zurich, Switzerland, 1998.