

# High-Performance, Data-parallel Document Inversion for the Cell/B.E. Processor

Daniele Paolo Scarpazza  
IBM T.J. Watson Research Center  
Multi-core Computing Department  
Yorktown Heights, NY 10598, USA  
dpскарpazza@us.ibm.com

## ABSTRACT

Text indexing is computationally expensive. Commercial search engines attack the task with massive, scalable, cluster-based solutions. But different domains (e.g., desktop, embedded, network appliances) are not compatible with a cluster solution. These domains would greatly benefit from a small form-factor, high-performance text indexing solution. Such a solution is the key enabler for new applications like wire-speed traffic indexing for network security forensics.

The Cell/B.E. Processor is a popular multi-core platform that promises enough compute power to perform live indexing, but its cores are notorious for architectural peculiarities (scratchpad memories, weak branch prediction) that require radical algorithm redesign to achieve acceptable performance. No previous work has investigated the potential of the Cell processor for indexing tasks.

In this work we consider document inversion, a core component of text indexing, and propose Blocked Hash-Based Inversion (BHBI), a data-parallel, single-pass, hash-based, in-core algorithm that maps well to the peculiarities we mentioned above.

We show the viability of our approach with a proof-of-concept implementation optimized for the Cell processor. Our tests show that our parallel document inversion is 1,200× faster than a single-core vanilla implementation of traditional Single-Pass In-Memory Inversion (SPIMI).

## 1. INTRODUCTION

The need for indexing is growing rapidly. In November 2004, Google reported 8 billion indexed pages. The volume of public and personal digital material we produce per year will grow between 2006 and 2010, from 161 to 988 billion Gbytes [12]. The web is dynamic, and it requires frequent revisitation and reindexing. Search engines are now part of common desktop applications. The need for fast indexing is appearing in new domains, like network security forensics, where analysts are contemplating the idea of performing live network traffic indexing, which requires impressive computational power.

Commercial search engines solve the problem by raw force: they use large clusters of commodity machines, in conjunction with a programming model (such as MapReduce [5]) that hides the complexity of domain decomposition, communication, synchronization and fault tolerance.

But architectures are changing quickly, and indexing algorithms are falling behind. For 30 years, a constant increase in clock frequency and architecture complexity has provided grow-

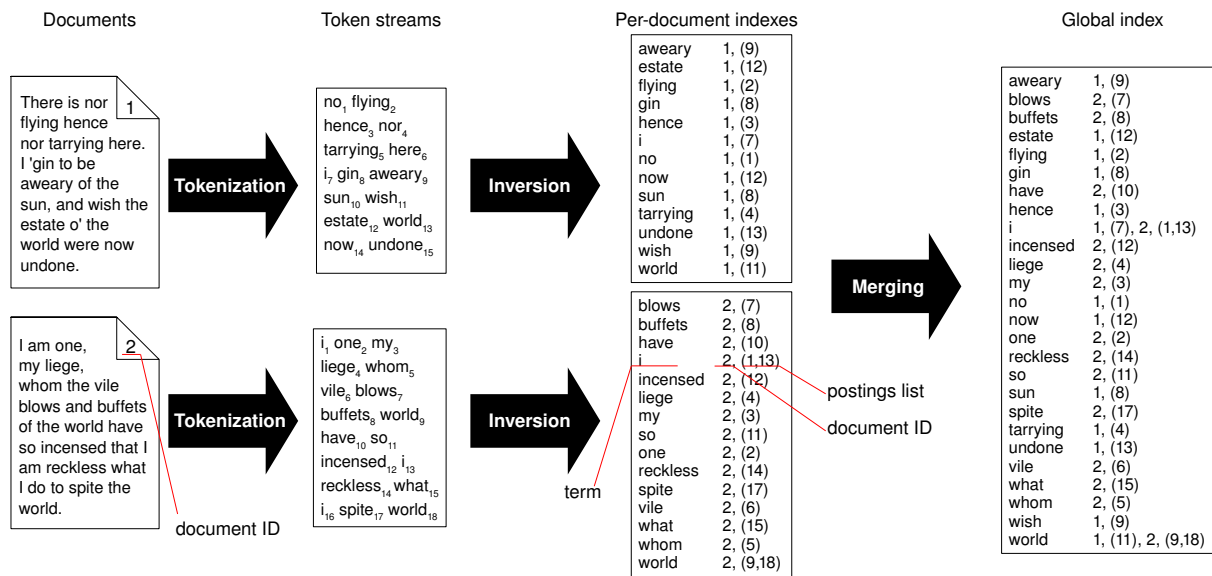
ing compute power per chip, at no additional effort for the application developer. This growth has come to an abrupt stop around year 2004. Architects now find better design alternatives by using more and simpler cores per chip. Complex branch prediction, branch speculation, and out-of-order execution may quickly become luxuries of the past because of area, power and heat constraints. The trend is towards more, simpler, in-order cores, with wider SIMD units, simpler branch prediction and no hardware speculation. Coherent caches and uniform memory spaces might disappear soon or suffer performance penalties. Core-local memories are small, and getting farther away from the main memory.

An example of these trends is IBM's Cell Broadband Engine [15] (the Cell, for short) with its 9 cores, 8 of which, called Synergistic Processing Elements (SPE) [8], are in-order, statically scheduled and cache-less, with 128-bit-wide SIMD units. The SPEs have a simple branch predictor and a high misprediction penalty, and no vector load/stores. Intel's Larrabee [20] will push some of these traits even further with 16, 32 or more in-order cores, featuring 512-bit-wide SIMD units and vector load/s/stores. Larrabee will still have dynamic branch predictors and a cache-coherent design, but it will support explicit cache control instructions to allow software management of the working sets.

While it is relatively easy to adapt current algorithms to expose thread-level parallelism, it is not trivial to adapt algorithms so that they use the increasing amount of data-level parallelism made available by wide SIMD units. Nor it is easy to employ explicit working sets that adapt to the memory hierarchy on board the hardware.

Despite the strong need for indexing and the popularity of multi-cores, little work has addressed the mapping of document inversion onto multi-cores. Authors focused on in-main-memory index merging [3], on index re-orderings [7] that reduce querying latencies, or on compressed text databases [18] that reduce footprint, all at the expense of indexing speed. No work addresses indexing with reference to multi-cores and data-level parallelism.

Our work addresses this gap, focusing on lightweight document inversion for multi-core processors. We radically rethink the algorithm in data-parallel terms, and we reduce the workload to a set of instructions that take only 25 clock cycles per token. This tight budget represents a significant technical challenge, and only allows for algorithms that map to a selected handful of instructions. The paper emphasizes algorithmic choices (e.g., term hashes in place of term identifiers) and implementa-



**Figure 1: The conceptual stages of a search engine’s indexer illustrated on an example. The tokenizer splits the input texts into a stream of tokens. The inverter generates a per-document sorted index, where entries associate a term with its postings. A final merge phase generates a global index, with postings coming from all documents.**

tion details (e.g., aligned, register-wide, single-payload storage scheme) that allow high efficiency with a basic set of indexing features, rather than focusing on advanced features like phrase searching, approximate matching, or proximity queries.

To the best of our knowledge, we are the first to propose a SIMDized document inversion algorithm. The contributions of this article are:

- a linear-time, hash-based, single-pass, document inversion algorithm that maps well on current multi-core platforms, and future ones with wider SIMD data parallelism, at the cost of a negotiable rate of search-time false positives);
- showing how low-level architectural issues have a strong, bottom-up influence on the design of algorithms;
- a proof of concept consisting of an optimized implementation for the Cell architecture, and a detailed performance analysis;
- a demonstration that the Cell processor is a valuable competitor on text-based workloads that differ significantly from the regular, array-based numerical workloads for which the processor was primarily designed.

We also provide a degree of document inversion performance (8.8 Gbyte/s per processor of throughput) that enables new applications previously considered computationally unfeasible: e.g., real-time traffic indexing for network security forensics.

## 2. DOCUMENT INVERSION

A search engine primarily performs two tasks: indexing and searching. Indexing is the construction of the index of a collection of documents. Searching takes this index and a query

as inputs, and produces a ranked list of documents as an output, where the documents match the query (according to some document model) and are sorted by decreasing relevance.

At a conceptual level, indexing is often divided into tokenization, document inversion, and index sorting. Figure 1 illustrates these tasks with an example.

*Tokenization* is the division of the input text into words, usually on the basis of a set of regular expressions that describe natural language words, URLs, e-mail addresses, dates, acronyms and company names, etc. Often, some linguistic processing takes place, like case folding, removal of stopwords (i.e., common words like “a”, “the”, “with”, ... that don’t help in judging how well a document matches a query), removal of dots from acronyms and genitive forms, stemming, compound word splitting, etc.

*Document inversion* transforms the sequence of tokens into a document index. This index enumerates, for each term, all the positions where it appears in the document. The process is called “inversion” to emphasize the fact that an index maps each term to its positions inside documents, and this is the inverse function of a document: a document conceptually maps a sequence of positions (1, 2, 3, ...) to the terms that appear there.

*Merging* fuses together the many document indexes to form a single global index, which contains all the terms that appear in the collection, and the positions of their occurrences, sorted by document ID and by word number (the rightmost box in Figure 1).

Figure 1 suggests a clean, well defined architecture with distinct, sequential phases. This clean separation of tasks is not representative of practical implementations: for data locality reasons, implementations often mix tokenization, inversion and sorting at different granularities (i.e., blocks of a document) to maximize data locality. We discuss our organization in Section 4.

### 3. OUR INVERSION ALGORITHM

Inversion algorithms have been traditionally aware of the huge latency gap between main memory and disk. Blocked Sort-Based Indexing (BSBI, see Algorithm 1) was designed to invert one block of input at a time and to produce one block of output, in main memory. This allows indexing documents arbitrarily larger than the available main memory and transfers large, sequential blocks that favor disk performance.

---

**Algorithm 1** Blocked sort-based indexing (BSBI). The algorithm stores inverted blocks in files  $f_1, f_2, \dots, f_n$  and the merged index in  $f_{merged}$ . Adapted from Manning et al. [16].

---

```

n ← 0
while all documents have not been processed do
  n ← n + 1
  block ← TokenizeNextBlock()
  BSBI-Invert(block)
  WriteBlockToDisk(block, fn)
end while
MergeBlocks(f1, f2, ..., fn; fmerged)

```

---

On multi-cores, one additional such gap demands attention: the latency gap between core-local memories and main memory. For example, on the Cell’s SPEs, a load access to the local store costs 6 clock cycles, while a main memory load costs up to ~830. On the POWER6 [21] architecture, loads cost 2 cycles from the L1 cache, 20–26 cycles from the L2 cache (still core-private), and ~450 from main memory. On the Intel Itanium family, loads cost 2, 6, 22 and ~200 cycles from L1, L2, L3 caches, and main memory, respectively. These large gaps encourage the design of algorithms that can keep their entire working sets in core-local memories.

BSBI is not suitable for multi-cores, because it maintains a term-termID dictionary that won’t fit the small core-local memories for any corpus of interesting size. Single-Pass In-Memory Inversion (SPIMI [10], Algorithm 2) overcomes this constraint.

---

**Algorithm 2** Inversion of a block in Single-Pass In-Memory Indexing (SPIMI) [10].

---

```

1: while ¬empty(token_stream) do
2:   output_file ← NewFile()
3:   dictionary ← NewHashTable()
4:   while free memory available do
5:     token ← GetNextToken(token_stream)
6:     if token ∈ dictionary then
7:       postings_list ← AddToDict(dictionary, token)
8:     else
9:       postings_list ← GetPostingsList(dictionary, token)
10:    end if
11:    if full(postings_list) then
12:      postings_list ← DoublePostingsList(dictionary, token)
13:    end if
14:    AddToPostingsList(postings_list, docID(token))
15:  end while
16:  sorted_terms ← SortTerms(dictionary)
17:  WriteBlockToDisk(sorted_terms, dictionary, output_file)
18: end while

```

---

SPIMI can index collections with an arbitrarily large vocabulary. It dispenses with termIDs, and uses a hash table to store the terms and their associated postings lists. Only the terms used in the current block appear in the table. Whenever there is no space left in the table, or no memory left to grow the lists, SPIMI commits its output block and starts a new one with an empty dictionary. Thanks to this property, SPIMI scales down to core-local memories.

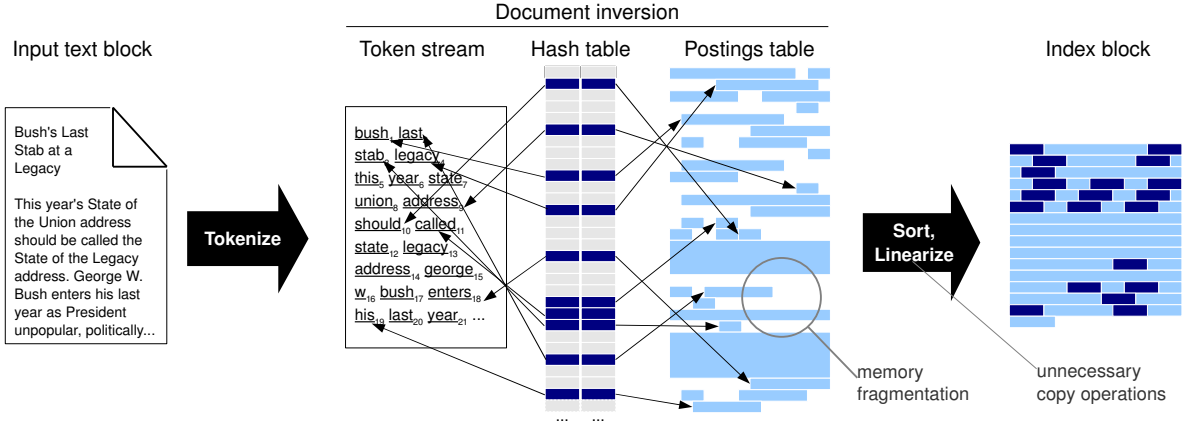
Nevertheless, SPIMI exhibits performance disadvantages that make it unfit for high-throughput applications. It allocates postings lists dynamically (line 7), and it reallocates them when full (line 12). This causes fragmented data structures (in the middle of Figure 2) and wasted memory. Plus, reallocation is expensive and requires unnecessary copy operations. SPIMI’s hash table can be implemented with *probing* or *chaining*: both have strong disadvantages (respectively, unpredictable access latencies and additional maintenance and traversal costs). SPIMI’s postings tables must be sorted by term, and written out in a linear fashion (as opposed to their fragmented layout in local memories). This linearization is expensive and requires unnecessary copy operations. Finally, SPIMI has a complex control flow that prevents SIMDization. Moreover, multi-cores like the Cell suffer strong performance penalties on branch-intensive code (24–26 cycles per mispredicted branch). Büttcher and Clarke [4] have explored space-performance trade-offs in postings list construction, but no traditional postings lists can be implemented on the Cell within a tight clock cycle budget.

To address these issues, we propose Blocked Hash-Based Inversion (BHBI) (Algorithm 3). BHBI is simpler than BSBI or SPIMI. Input and output buffers are pre-dimensioned to use local memories efficiently, avoiding dynamic allocation and excluding overflows. BHBI takes one token  $t$  from the input (the  $w^{\text{th}}$  word in document  $docID$ ) and adds a corresponding entry ( $hash(t), docID, w$ ) to the output block. When the output block is complete, it sorts all the entries by field content (the sorting key is the entire entry), and it writes the block to main memory.

BHBI uses a storage scheme called *single payload* (in the terminology of Melnik et al. [1]) because each entry in the output table represents one token only. BHBI does not explicitly construct postings lists: rather, postings lists emerge later, after the single-payload entries are sorted. Multiple occurrences of the same term  $t$  appear as a sequence of entries ( $hash(t), docID, i_1$ ), ( $hash(t), docID, i_2$ ),  $\dots$ . The task of compacting the index into a more space-efficient format, such as ( $hash(t); docID_i, (i_1, i_2, \dots); docID_j, (j_1, j_2, \dots)$ ) is optional, postponed to the index merge task, and we do not discuss it here.

BHBI uses no term dictionary. It extracts a hash of each token, which it uses as a term identifier (e.g., as in Dmitriev et al. [6]). Note that this usage is not a hash table. To avoid confusion, note that hashes are stored and sorted, rather than used as indexes in a hash table. Terms are represented by their hashes in the output and in any stage that follows inversion. The association between a hash and its term is deliberately not stored anywhere.

These radical design choices have the advantages that follow. There is no dynamic memory allocation and reallocation, and no consequent unnecessary copy operations. Explicit buffer management allows the working set to fit in small core-local memories, and to employ double buffering. Each iteration of the inversion kernel loop (lines 4–6) processes independent inputs and produces independent outputs, and it is free from loop-carried



**Figure 2: Single-Pass In-Memory Inversion (SPIMI) and its drawbacks: (1) the dynamically (re-)allocated postings lists cause memory fragmentation, (2) list reallocation require undesired copy operations, (3) the need to linearize the lists before emitting them to output causes further undesired copy operations.**

**Algorithm 3** Blocked Hash-Based Inversion (BHBI), the inversion algorithm we propose. Variables marked as ‘locals’ are intended to be allocated in core-local memory, while input and output data are in main memory.

Constants:  $B$ , size of each of the local buffers;  
Input:  $docID$ , document identifier;  
 $T = (t_1, t_2, \dots, t_N)$ , tokens in the input block;  
Output:  $X = (x_1, x_2, \dots, x_{\lceil N/B \rceil})$ , output blocks;  
Locals:  $w$ , document-relative position of the first document word in the buffer;  
 $TB = (tb_1, tb_2, \dots, tb_B)$ , token input buffer;  
 $XB = (xb_1, xb_2, \dots, xb_B)$ , output buffer;

The symbol ‘ $\leftarrow$ ’ denotes a variable assignment in local memory.  
The symbol ‘ $\Leftarrow$ ’ denotes a transfer from/to main memory.

```

1:  $w \leftarrow 0$ 
2: while  $w < N$  do
3:  $(tb_1, tb_2, \dots) \Leftarrow (t_{w+1}, t_{w+2}, \dots, t_{\min(w+B, N)})$ 
   /* load a block of input token into the token buffer */
4: for all  $tb_i \in TB$  do
5:  $xb_i \leftarrow (\text{hash}(tb_i), docID, w + i)$ 
6: end for
7:  $w \leftarrow w + B$ 
8:  $(xb_1, xb_2, \dots, xb_B) \Leftarrow \text{Sort}(xb_1, xb_2, \dots, xb_B)$ 
9:  $x_i \leftarrow (xb_1, xb_2, \dots, xb_B)$ 
   /* store a block of output entries to main memory */
10:  $i \leftarrow i + 1$ 
11: end while

```

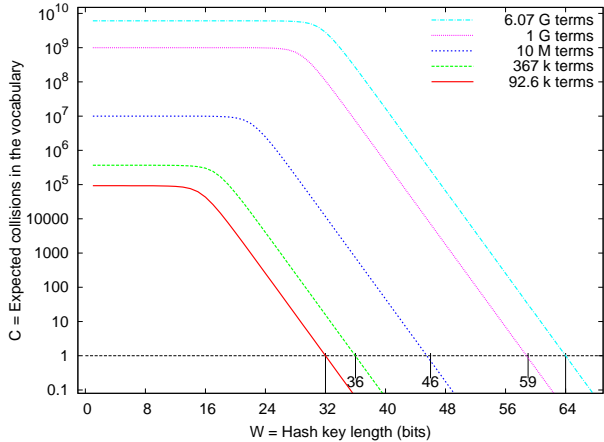
dependencies and SIMDizable/unrollable *ad libitum*. There is no hash table, and no expensive probing. Expensive variable-length string manipulations are replaced with inexpensive hashes; this also speeds sorting, since comparing aligned entries costs much less than variable length strings, as Inoue et al. [13] show.

BHBI also allows three additional optimizations whose impact on performance depends on the target platform:

- input and the output entries can be aligned to the register width; aligned loads and stores, in general, cost less than misaligned ones. SPIMI’s output format, with its variable-length terms, is more expensive to generate;
- by matching the size of input and output entries, BSBI can operate *in place*, saving memory. By *in-place* we mean that the algorithm rewrites input locations with output data without needing extra memory; this is a valuable feature, considering the small size of in-core memories;
- a later sort step can assume that entries with the same hash are already in the right order by construction. I.e., entry  $(h, docID, w + 1)$  necessarily appears after  $(h, docID, w)$ . Our implementation takes advantage of all of the above, as described in Section 4.

BHBI’s main limitations are: (1) the index no longer contains plaintext terms, and (2) hashes can collide. (1) is acceptable, because the index contains references to the tokens in the original document. (2) is more serious: when terms  $t_1$  and  $t_2$  collide, postings of the two terms mix in the same list during indexing. This leads to false positives at query time and impacts scoring/ranking performed with TF-IDF [14]. By virtue of the *birthday problem*, the expected number of conflicts  $\bar{C}$  incurred by a *good*  $W$ -bit wide hashing function<sup>1</sup> over a vocabulary of  $M$  terms behaves as in Figure 3. If we assume that  $\bar{C} < 1$  is safe, then the maximum safe  $M$  depends on  $W$  as follows:

<sup>1</sup>A hash function of size  $W$  bits is *good* if, applied on a vocabulary, returns a sequence of independent, identically distributed uniform random variables over the range  $[0, 2^W - 1]$



**Figure 3:** The expected number of term hash collisions  $\bar{C}$  in a hashed term dictionary is a function of the hash width  $W$ .

$$M < \frac{1 + \sqrt{1 + 8 \cdot (2^W - 1)}}{2} \approx \frac{1}{2} + 2^{\frac{W}{2} + \frac{1}{2}}.$$

A 32-bit hash is safe up to 92,682 terms; the Kiwix Wikipedia dump [2] (367,277 terms) needs at least 36 bits; dictionaries of 10 million and 1 billion terms require 46- and 59- bit hashes, respectively. Our implementation employs a 64-bit hash, which is safe up to 6.074 billion distinct terms. For larger collections, our approach can adapt to 128-bit or wider hashes with no loss of generality.

The size of a collection in tokens  $T$  correlates with the number of distinct terms  $M$  according to Heaps' Law [9]  $M = k \cdot T^b$ , plotted in Figure 4 for a few relevant corpora. The Kiwix corpus fits the law with  $b = 0.68$  and  $k = 5.05$ . Consequently, the maximum size of a Wikipedia-like collection safely indexed with a 64-bit hash according to Heaps' Law is  $4.83 \cdot 10^{13}$  tokens = 422.42 Terabytes = or 7.2 billion avg. Wikipedia pages.

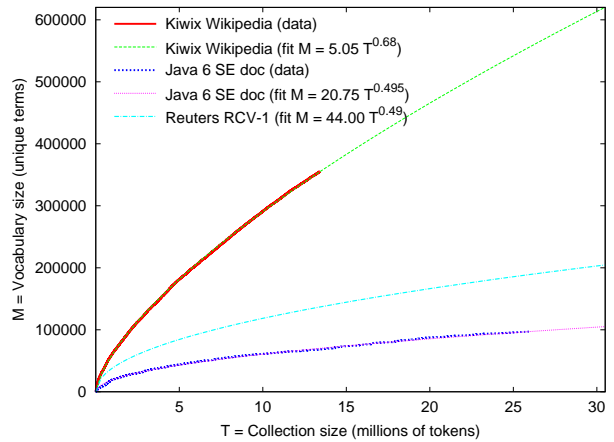
## 4. EXPERIMENTAL RESULTS

We present a proof-of-concept implementation of our BHBI document inverter on the Cell processor. This stage is deeply optimized and achieves high performance. With the additional intent of proving the general feasibility of our approach, we have also implemented an end-to-end text indexer that employs our BHBI inverter. Our implementation runs on the PowerXCell8i processor running at 3.2 GHz, as available on the IBM QS22 blades<sup>2</sup>.

Our implementation comprises four stages:

- **stage (1):** a finite-state-machine-based tokenizer specified by the same regular expressions that appear in Lucene's tokenizer; this stage is parallelized and SIMDized; its output is a token table as described below;

<sup>2</sup>We employed the Cell SDK version 3.0, which includes the GNU GCC compiler version 4.1.1. We wrote our code in C with SPU intrinsics [11]. Intrinsics allow precise instruction-level control without incurring the time-consuming manual tasks of assembly programming (e.g., register variable allocation).



**Figure 4:** Heaps' Law correlates the vocabulary size  $M$  and the collection size  $T$ . Collections include the Kiwix Wikipedia [2], Java 6 documentation, Reuters' RCV1 corpus [16]). Thick lines are actual data; thin lines are Heaps'-Law fits.

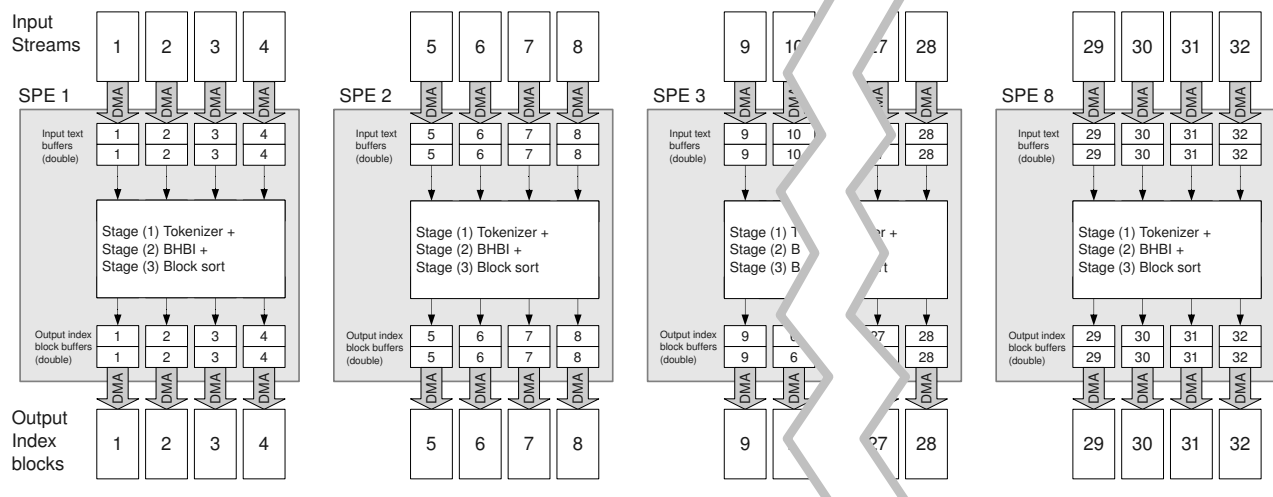
- **stage (2):** a BHBI document inverter based on Algorithm 3, fully optimized to take advantage of SIMD instructions; this stage generates token-hash tables where entries appear in the same order as in the text;
- **stage (3):** a block sort stage; this stage sorts the contents of each output index block by hash value, before committing it into main memory;
- **stage (4):** a global merge sort that fuses together all the block indexes into a global one.

This section is intended to show the viability and the performance of stage (2) in a realistic use case. Our implementation of stages (1), (3) and (4) does not necessarily contain novel contributions.

Stages (1), (2) and (3) operate in-core, and they are coupled in a streaming fashion so that they maximize data locality: a single block of input text from a single document is read from main memory (DMA) into the local store; it is then tokenized, inverted and sorted, and it produces a single sorted output index block that is written back to main memory (the second DMA). Double buffering ensures that DMA operations take place at no cost while the SPEs are busy with computation. Stage (4) operates on out-of-core main memory, it is separate from the previous stages and it employs a tournament-tree merge sort, assisted by radix histograms, as detailed later.

Stage (1) is fundamentally a finite state machine, sequential in nature. A single machine does not expose sufficient data-parallelism to exploit the hardware efficiently. To that purpose, we use a tokenizer [19] that aggregates 4 finite-state machines into single flow of SIMD instructions.

A single iteration of steps (1)–(3) consists of four data-flows happening in parallel on each SPE. Since there are 8 SPEs on each Cell chip, a minimum of  $4 \times 8 = 32$  input documents are needed at any time to keep our algorithm at peak throughput.



**Figure 5: The software architecture we adopt for stages (1)–(3) in our proof-of-concept implementation on the Cell processor.**

Stage (2) is implemented as in Algorithm 3 with the exception of DMA loads (line 3 in the listing). These transfers are not required because the tokens are already in the local store after tokenization.

An iteration of stages (1)–(3) processes four input text blocks, each 4kbyte in size, and generates four corresponding output index blocks, using a single-payload storage scheme. With Wikipedia-like input, each output block contains (on average) 553.25 tokens, which makes the cumulative size of data written back to memory 35,408 bytes.

The input of Stage (2) is a token table where entries are aligned to the register width (16 bytes = 128 bits). Each entry has four 32-bit fields: pointers to the beginning and the end of the token within the input buffers, a token type code (useful to perform linguistic processing), and padding. BHBI reads one such line, loads the string into registers (taking care of alignment and truncating after the 16<sup>th</sup> byte), computes the hash with fast, SIMDized code, and saves back one line, in place. The output line has the same 128-bit width, and three fields: a 64-bit hash, the document ID, and the word number of the token.

Stage (2) as just described can be optimized very well on the Cell, especially thanks to the fact that entries are aligned and independent from each other. We use a SIMDized hash function, SIMD string manipulation and loop unrolling, achieving a  $28.91\times$  speedup with respect to a naïve implementation based on C library string functions and a vanilla DJB2 hash implementation, as Figure 6(a) shows. We also achieve a very high resource utilization, e.g., our CPI is 0.75, as in Figure 6(c). This is close to 0.5, the ideal CPI for the SPE<sup>3</sup>.

Stage (3) sorts the output block before storing it to main memory. For stage (3), any sorting algorithm is a valid choice. The design of optimized sorting algorithms on the Cell processor is an open problem, and discussing it is beyond the purpose of this paper. We only report the details of our practical implementation necessary for the reader to understand our approach. Asymptotic analysis provides little help in the design of sorting algorithms

on the Cell because: (1) small amounts of data are processed at a time, due to the paucity of the core-local memories, and (2) high misprediction penalty taxes heavily those algorithms that have a complex control flow.

We employ a block sort implementation that runs, on the average, twice as fast as quicksort, as Figures 6(a,b) show. This algorithm is *American flag sort* [17], a linear-time *histogram sort* variant that distributes items into hundreds of buckets. The first step counts the number of items in each bucket, and the second step computes where each bucket will start in the array. The last step moves items to their proper buckets. We only perform one run of the sort with 256 buckets, on the basis of the first byte of the key (the hash). Then, we run an unrolled, SIMDized *bubble sort* on each of the bins. Despite its high asymptotic cost ( $O(N^2)$ ), bubble sort runs fast on the small numbers of per-bucket entries, and it is amenable to unrolling and SIMDization. Results are promising, but the algorithm is still control-intensive and suffers from branch miss stalls. Further research on this sort is beyond the scope of this work.

Stage (4) is a large scale multi-way merge sort, where the input and the output together might occupy the entire available main memory. In our tests, a tournament-tree sort turned out to be the fastest implementation for this work. For the sake of simplicity, our implementation does not consider the extension of this merge sort to disk storage. Stage (4) employs radix histograms to balance the load across the SPEs: we divide the population of a run into bins, depending on the prefix of each key. We consider 8-bit prefixes, and therefore 256-bin histograms. In the later stages of merge sort, when the number of available SPEs (8 per chip, 16 on a QS22 blade) exceeds the number of runs to sort, we partition the radix domain (0...255) into as many partitions of contiguous values as the SPEs, balancing the size of each partition. With this method, each partition does not overlap or interact with adjacent ones, and therefore each can be sorted independently. The memory coordinates of each partition can be determined exactly a priori, by summing values in the histograms.

Table 1 shows the performance of our indexer on Wikipedia

<sup>3</sup>the ideal CPI is 1/2 because SPEs have two pipelines and can issue two instructions in the same clock cycle, provided that they do not conflict.

Optimization	CPI	Cycles/ token	Mtoken/s (1 SPE)	Gbyte/s (1 SPE)	Speedup
(1) Sequential, DJB2 hash	1.69	723.46	4.423	0.039	=1.00×
(2) SIMD Hash Function	1.97	249.86	12.807	0.112	2.90×
(3) SIMD String Realign	1.72	67.06	47.720	0.417	10.79×
(4) Unroll ×2	0.98	34.82	91.912	0.803	20.78×
(5) Unroll ×4	0.79	26.90	118.949	1.040	26.89×
(6) Unroll ×8	0.75	25.02	127.889	1.118	28.91×

Figure 6(a): Each significant optimization we applied (a SIMDized hash function, SIMD string manipulation, loop unrolling) achieves a  $\sim 3\times$  speedup.

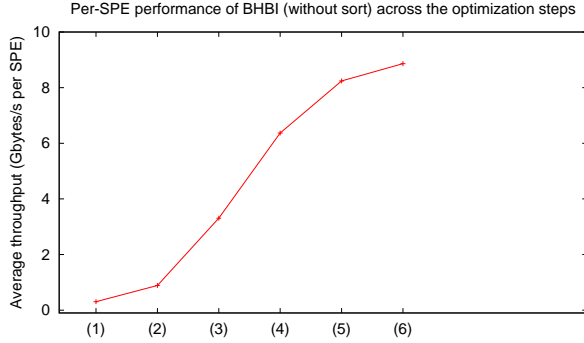


Figure 6(b): Average throughput per SPE, in Gbyte/s of processed input text.

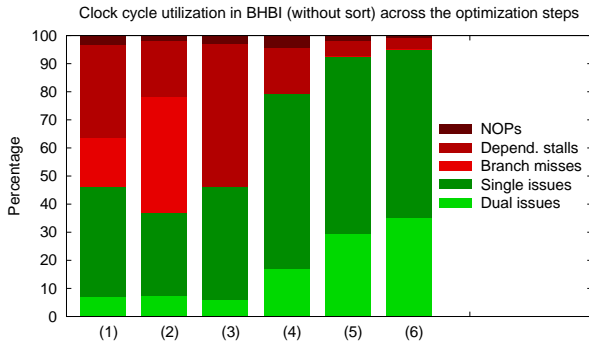


Figure 6(c): Utilization of the clock cycles across the optimization steps. Shades of green indicate productive cycles (single or dual issues); shades of red indicate unproductive ones. The chart shows that our optimizations get rid of the branch misses, a significant portion of dependency misses, and achieve a good rate of dual issues.

**Figure 6: (a),(b),(c): Performance of BHBI (without sorting) across the optimization steps.**

data. BHBI inversion turns out to be by far the least expensive task, absorbing less than 2% of the entire execution time, completing a new token on each core every 25 clock cycles, and every 3.13 cycles on one entire Cell chip, which is a remarkable result.

## 5. DISCUSSION

We compare the performance of our optimized BHBI approach against a single-core vanilla SPIMI implementation. The results are in Table 2. The comparison against SPIMI running on the PPE of the Cell processor is relevant, because this is the level of performance that designers would obtain if they didn't employ parallelization and optimization. Sub-tasks like “invert”, “sort”, and “linearize” have been re-grouped to make the comparison fair (BHBI does not have an explicit linearize phase).

Optimization	CPI	Cycles/ token	Mtoken/s (1 SPE)	Mbyte/s (1 SPE)	Speedup
(1) Merge sort	1.53	425.15	7.53	65.78	0.75×
(2) Quicksort	2.36	320.35	9.99	87.31	=1.00×
(3) Histogram + Bubble	1.89	189.27	16.91	147.77	1.69×
(4) Unroll ×4 histogram	1.74	163.92	19.52	170.62	1.95×
(5) Unroll ×4 bubble	1.75	156.14	20.50	179.13	2.05×

Figure 7(a): The sorting implementation we optimized for the purpose of this paper achieves a  $\sim 2\times$  speedup with respect to quicksort. Still, resource utilization is improvable, as indicated by the relatively high CPI.

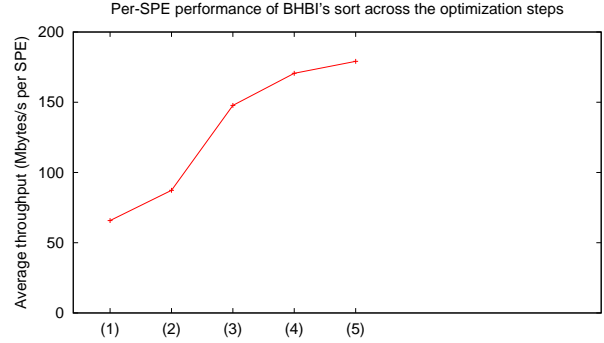


Figure 7(b): Average throughput per SPE, in Mbyte/s of input text processed.

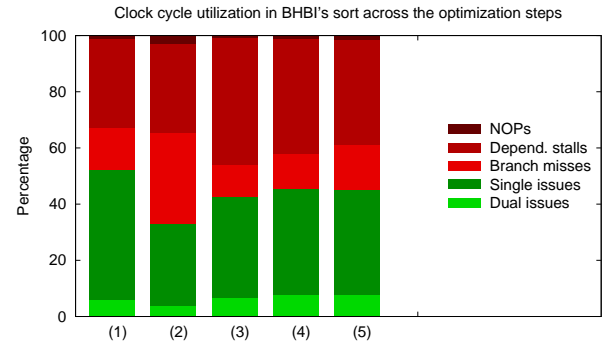


Figure 7(c): Utilization of the clock cycles across the optimization steps. Shades of green indicate productive cycles (single or dual issues); shades of red indicate unproductive ones. We achieve a significant overall speedup, but not a significant increase in resource utilization.

**Figure 7: (a),(b),(c): Performance of the sort used by BHBI across the optimization steps.**

Our multi-core BHBI running on the 8 SPEs of a Cell processor turns out to be 208.93× faster than the single-core, non-parallelized SPIMI. Even assuming that SPIMI could be ported onto the 8 SPEs and enjoy an 8× speedup, BHBI would still have a 26× advantage.

For the convenience of the reader, the table also includes the throughput values of tokenization and index merging, even though they are out of scope for this paper. For the purpose of comparison against a traditional single-core processor, the last column reports SPIMI performance on the Intel Pentium 4 processor.

All the algorithms we propose enjoy the desirable property of being SIMDizable to larger widths. This paves the way for further performance gains on future multi-core generations. Our algorithm has reduced the relative contribution of document in-

	Stage (1) Tokenization <sup>1</sup>	Stage (2) BHBI Inversion	Stage (3) Block sort <sup>1</sup>	Stage (4) Global Merge-sort <sup>1</sup>	Cumulative end-to-end
<b>Latency</b>					
Clock cycles per block <sup>2</sup>	491,605	<b>55,548</b>	347,330	1,926,137	2,820,620
Latency per block* ( $\mu$ s)	153.63	<b>17.36</b>	108.54	601.92	881.44
Cycles per token / SPE	221.44	<b>25.02</b>	156.45	867.63	1,270.55
Cycles per token / Cell chip	27.68	<b>3.13</b>	19.56	108.45	158.82
Relative contribution	17.43%	<b>1.97%</b>	12.31%	68.29%	100.00%
<b>Throughput</b>					
Millions of tokens per second (1 SPE)	14.00	<b>123.89</b>	19.81	3.57	2.44
Millions of tokens per second (1 Cell)	111.99	<b>991.14</b>	158.51	28.58	19.52
Avg. Wikipedia articles <sup>3</sup> per second (1 SPE)	2,087.58	<b>18,475.26</b>	2,954.72	532.81	363.84
Avg. Wikipedia articles <sup>3</sup> per second (1 Cell)	16,700.62	<b>147,802.05</b>	23,637.77	4,262.47	2,910.75
Consumed input Mbyte/s (1 SPE)	122.35	<b>1,082.82</b>	173.17	31.23	21.32
Consumed input Mbyte/s (1 Cell)	978.81	<b>8,662.59</b>	1385.4	249.82	170.6

<sup>1</sup>Tokenization and sorting are not the object of this paper; their performance is reported solely for completeness.

<sup>2</sup>By “block” we mean the computation required to process 4 blocks of 4kbyte-sized text input buffers per SPE, 32 blocks in total.

<sup>3</sup>Our throughput estimates assume an average token of 8.74 bytes and an average article of 6705.8 tokens. Measurements by the authors.

**Table 1: Performance obtained by our proof-of-concept BHBI-based indexer on Wikipedia-like data.**

version to a negligible percentage (<2%) of the entire text indexing workload.

Task	SPIMI PPE only (Mbytes/s)	BHBI, etc. 8 SPEs (Mbytes/s)	Speedup (BHBI vs.) SPIMI	SPIMI Intel P4 3.2GHz (Mbytes/s)
Tokenization	19.82	1,001.25	50.52×	(49.67)
Document inversion: invert & linearize	5.85	1,221.86	<b>208.93×</b>	(13.48)
block sort	6.86	8,862.59	<b>1,291.73×</b>	(15.68)
Global mergesort	39.62	1,385.54	34.97×	(96.06)
	30.75	249.82	8.12×	(32.50)

**Table 2: Performance comparison between our BHBI implementation and our best SPIMI implementation.**

## 6. CONCLUSIONS

We analyzed traditional document inversion algorithms like BSBI and SPIMI, and showed that they map poorly to multi-cores. This is due to their large working sets that do not fit small core-local memories, to their expensive, space-inefficient postings lists, and to their complex control flow that hinders an efficient utilization of wide SIMD units.

We propose a new inversion technique called BHBI (Blocked Hash-Based Inversion) that overcomes these limitations. BHBI operates on small core-local memories, employs a low-maintenance single-payload storage scheme to collect the postings, and benefits from a growth in SIMD width.

As a proof of concept, we provide an implementation of BHBI that exploits the degrees of parallelism available on the Cell processor, achieving an inversion throughput of 1.22 Gbyte/s, i.e., more than 200× faster than a vanilla SPIMI implementation on the same hardware.

We recommend the use of BHBI in current and future, multi-core-based search engines.

## 7. REFERENCES

- [1] Building a distributed full-text index for the web. In *ACM Tenth Intl. World Wide Web Conf. (WWW10)*, Hong Kong, May 2001.
- [2] Kiwix, version 0.5, www.kiwix.org, Mar. 2007.
- [3] S. Bütcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *14th ACM Conf. on Information and Knowledge Management (CIKM’05)*, pages 317–318, 2005.

- [4] S. Bütcher and C. L. A. Clarke. Memory management strategies for singlepass index construction in text retrieval systems. Technical Report CS-2005-32, University of Waterloo, Canada, Oct. 2005.
- [5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symp. on Operating System Design and Implementation (OSDI’04)*, San Francisco, CA, Dec. 2004.
- [6] P. A. Dmitriev, N. Eiron, M. Fontoura, and E. Shekita. Using annotations in enterprise search. In *15th Intl. Conf. on World Wide Web (WWW’06)*, pages 811–817, New York, NY, USA, 2006. ACM Press.
- [7] S. Garcia, H. E. Williams, and A. Cannane. Access-ordered indexes. In *New Topological Descriptors. J. Chem. Inf. Comput. Sci. 1994*, pages 7–14. Zealand, 2004.
- [8] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell’s Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.
- [9] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., Orlando, FL, USA, 1978.
- [10] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Jnl. of the American Society for Information Science and Technology*, 54:713–729, 2003.
- [11] IBM. *PPU & SPU C/C++ Language Extension Specification, version 2.5*. IBM, Sept. 2007.
- [12] IDC Corporation. The expanding digital universe. *White Paper*, March 2007.
- [13] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *16th Intl. Conf. on Parallel Architecture and Compilers (PACT’07)*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.
- [14] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Jnl. of Documentation*, 28:11–21, 1972.
- [15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, pages 589–604, July/September 2005.
- [16] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008.
- [17] P. M. McIlroy, K. Bostic, , and M. D. McIlroy. Engineering radix sort. 6:5–27, 1993.
- [18] G. Navarro. Indexing text using the ziv-lempel trie. *Jnl. of Discrete Algorithms*, 2:2004, 2002.
- [19] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In *23rd Intl. Conf. on Supercomputing (ICS’09)*, Yorktown Heights, New York, USA, June 2009.
- [20] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008*, pages 1–15, New York, NY, USA, 2008. ACM.
- [21] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Flühr. Design and implementation of the POWER6 microprocessor. *Solid-State Circuits, IEEE JNL of*, 43:21–28, Jan. 2008.