

# Workload Characterization and Optimization of High-performance Text Indexing on the Cell Broadband Engine™ (Cell/B.E.)

Daniele Paolo Scarpazza  
IBM T.J. Watson Research Center  
Multicore Computing Department  
Yorktown Heights, NY 10598, USA  
dpскарпаzza@us.ibm.com

Gordon W. Braudaway  
IBM T.J. Watson Research Center  
Multicore Computing Department  
Yorktown Heights, NY 10598, USA  
braud@us.ibm.com

## ABSTRACT

In this paper we examine text indexing on the Cell Broadband Engine™ (Cell/B.E.), an emerging workload on an emerging multi-core architecture. The Cell Broadband Engine is a microprocessor jointly developed by Sony Computer Entertainment, Toshiba, and IBM (herein, we refer to it simply as the “Cell”).

The importance of text indexing is growing not only because it is the core task of commercial and enterprise-level search engines, but also because it appears more and more frequently in desktop and mobile applications, and on network appliances. Text indexing is a computationally intensive task. Multi-core processors promise a multiplicative increase in compute power, but this power is fully available only if workloads exhibit the right amount and kind of parallelism.

We present the challenges and the results of mapping text indexing tasks to the Cell processor. The Cell has become known as a platform capable of impressive performance, but only when algorithms have been parallelized with attention paid to its hardware peculiarities (expensive branching, wide SIMD units, small local memories).

We propose a parallel software design that provides essential text indexing features at a high throughput (161 Mbyte/s per chip on Wikipedia inputs) and we present a performance analysis that details the resources absorbed by each subtask. Not only does this result affect traditional applications, but it also enables new ones such as live network traffic indexing for security forensics, until now believed to be too computationally demanding to be performed in real time.

We conclude that, at the cost of a radical algorithmic redesign, our Cell-based solution delivers a 4× performance advantage over recent commodity machine like the Intel Q6600. In a per-chip comparison, ours is the fastest text indexer that we are aware of.

## 1. INTRODUCTION

The rate at which we globally produce new information [1] creates a strong need for fast information indexing. Not only new contents appear every day, but most contents are dynamic and require frequent re-visitation and re-indexing to keep indexes up to date.

These tasks, by today’s standards, demand a huge amount of computation. A technique to perform indexing inexpensively can impact not only traditional domains but also revolutionize new ones, like network security forensics. Analysts are contemplating live network traffic indexing, a concept that hasn’t turned into a practical implementation because of its computational cost.

With the multi-core revolution, processors are offering increasing amounts of parallelism: more and more Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP) in addition to Instruction-Level Parallelism (ILP). The number of cores per chip is growing, and

use of Simultaneous Multi-Threading (SMT) and SIMD units (Single Instruction, Multiple Data) is becoming more common.

Commercial search engines perform indexing on large clusters of processors employing a programming model (MapReduce [2]) that hides the complexity of domain decomposition, communication, synchronization and fault tolerance. Among the different forms of parallelism, DLP is the one that traditional programming models neglect the most. This causes a sub-optimal resource utilization in contemporary hardware, and an even worse utilization in future hardware that has wider SIMD units. Contemporary commodity hardware often has 128-bit wide SIMD units (i.e., the Intel Core 2 chips and the Cell [3] chip). Future architectures will feature 256-bit or 512-bit wide SIMD units, like Intel AVX [4] and Larrabee [5], respectively.

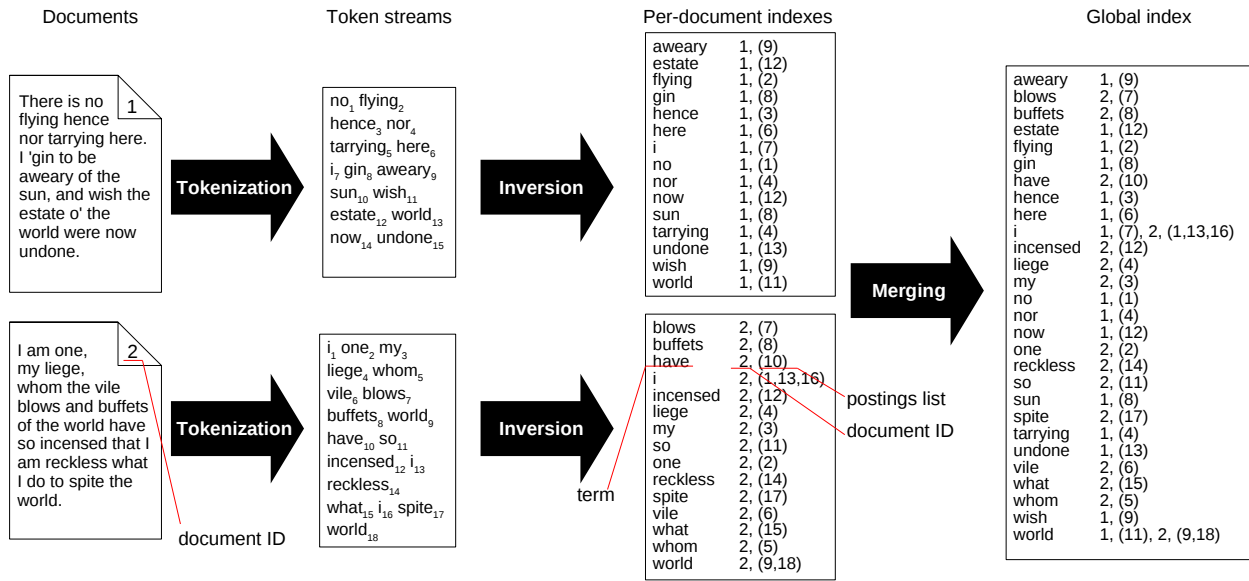
Our work addresses precisely this gap. We propose text indexing techniques that use all levels of parallelism, including DLP. We present and characterize an indexer software design for the Cell processor [3, 6, 7], a good example of a data-flow-oriented platform that offers a significant amount of DLP.

The Cell is a 9-core chip: a traditional PowerPC (called Power Processing Element, or PPE), and eight other worker cores called Synergistic Processing Elements (SPE [8]). The PPE and the SPEs have 128-bit wide SIMD units. The Cell has a deserved reputation of being difficult to program because of the spartan architectural features of its SPEs: small, non-coherent per-core scratchpad memories, and simple branch predictors with high misprediction penalties. The SPEs do not run irregular, control-dominated code well. On the other hand, the missing features that the SPEs lack are power-hungry (caches, dynamic branch prediction, hardware speculation<sup>1</sup>, cache-coherency, out-of-order execution). As a consequence, the Cell is the most power-efficient chip for high-performance computing. Cell-based machines occupy the first seven positions of the Green500 List [10]. RoadRunner [11], one of them, is currently the most powerful supercomputer in the world [12]. In light of these considerations, it still makes sense to redesign a workload like text indexing, that is traditionally composed of control-intensive, inherently sequential tasks, into a data-parallel model, despite the high development effort required.

The relevant contributions of this article are: a radical rethinking of text indexing tasks in terms of data-parallel algorithms that are amenable to a SIMD implementation; a proof-of-concept optimized implementation that achieves an impressive end-to-end throughput; and a correspondingly detailed performance characterization.

This paper is organized so that the most relevant results are summarized in Section 4. The following sections delve into the details of the subtasks that compose indexing, and they can be skipped at a first read.

<sup>1</sup>On the Pentium 4, an architecture with aggressive hardware speculation, 26%–45% of the decoded micro-ops are discarded [9].



**Figure 1: The tasks of search engine's indexer, illustrated on an example. The tokenizer splits the input texts in a stream of tokens. The inverter generates a per-document sorted inverted index. The merge phase generates a global index, with postings coming from all documents.**

## 2. TEXT INDEXING BASICS

We now provide an introduction to the fundamental concepts of text indexing for readers unfamiliar with information retrieval topics.

A search engine performs two tasks: indexing and searching. In general terms, indexing consists of constructing the index of a collection of documents; searching consists of using this index to produce a list of documents matching a user's query, sorted by decreasing relevance. For the purposes of this work, we focus exclusively on indexing.

At a conceptual level, indexing comprises three stages: tokenization, document inversion, and index sorting. Figure 1 illustrates these stages with an example. The documents at the left side of the figure represent the input collection; the black arrows are compute stages; the final output index is at the right side of the figure.

*Tokenization* splits the input text into tokens, such as natural language words, URLs, e-mail addresses, dates, acronyms, etc. Tokens are specified by regular expressions. Often, a tokenizers include linguistic processing like case folding, stemming, acronym normalization, compound word splitting, etc.

*Document inversion* transforms the sequence of tokens into a document's *index*. An index contains a sorted list of the terms contained in the document and, for each term, a list of the locations of its occurrences in the document. A common way to represent an occurrence is the word position of where the term appears in the document. In the figure, entry "i, 2, {1,13,16}" indicates that the word "i" appears in document 2 in word positions 1, 13 and 16.

The *merging* stage fuses the many document indexes into one sorted global index. This index has the same structure as the per-document indexes, except it represents tokens and occurrences from all the documents of the collection. Despite its simplicity, the global index can resolve efficiently a large spectrum of structured queries, either boolean or based on TF-IDF models (Term Frequency, Inverse Document Frequency [13]), including proximity constraints.

Figure 1 suggests an organization with distinct, sequential phases. This is not representative of practical implementations, as we show in Section 4. To preserve data-locality, it is much more convenient to

split computation into units that process smaller, fixed-size blocks of data.

## 3. TARGETING THE CELL PROCESSOR

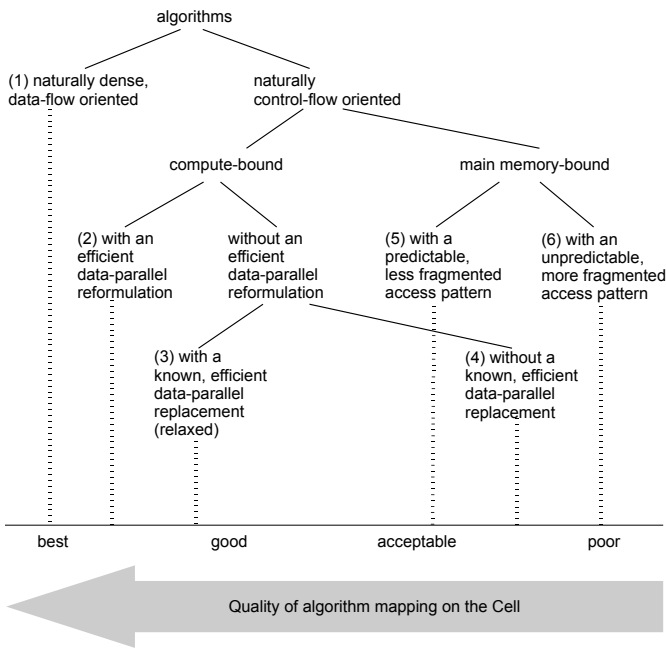
On a Cell processor, the PPE serves as a service processor, generally intended to run the operating system and coordinate SPE threads. To achieve high performance, workloads must exploit the SPEs. Designing efficient software for the SPEs is challenging because:

- branches are expensive (branch predictors are simple, and a misprediction costs 24–26 clock cycles);
- an SPE can operate directly only out of its small, local memory, called Local Store (LS), 256 Kbytes in size. SPEs have no cache memories. Programmers must design algorithms that use an explicit working set, small enough to fit the LS;
- Direct Memory Accesses (DMA) are required to transfer data from/to the main memory. Programmers must overlap computation with DMA transfers to hide the shorter of the two latencies;
- SIMD instructions operate on 128-bit registers. Programmers can use SIMD to process four 32-bit operands at the same time. This requires accurate data layout and, often, removal of control flow;
- compilers or programming frameworks offer little help in parallelizing and vectorizing code.

We divide algorithms into six classes to predict how well they can map to the Cell platform, as in Figure 2.

Class 1 contains naturally data-flow oriented algorithms (e.g., regular, numerical, dense array-based computations). They map well to the Cell, whether they are compute-bound or main-memory bound (depending on their *arithmetic intensity*; see the *Roofline* performance model [14]).

Class 2 contains compute-bound algorithms that are naturally expressed in a sequential form but can be transformed into a branchless,



**Figure 2: A taxonomy of algorithms, and the expected quality of their mapping to the Cell platform.**

SIMD, data-parallel form with some effort. Central to this transformation is the replacement of branches with software-level speculation. By “software-level SIMD speculation” we mean, for example, code that uses data-parallel compare instructions followed by alteration of data-parallel variables based on the results of the compare. In this manner, we accomplish the conditional modification of individual values within a data-parallel variable without the use of branch instructions.

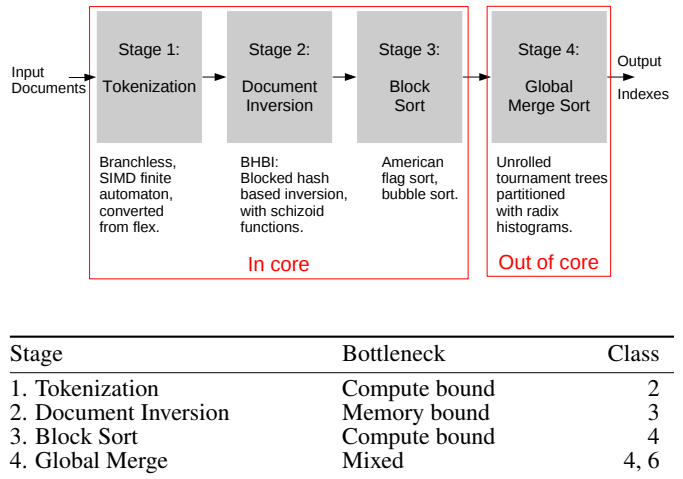
If the speculated portion is not too large, the transformed code runs much faster than the original. Branch stalls are virtually eliminated, and SIMD instructions can yield a significant speedup. Many text stream-based algorithms are amenable to this transformation, including the tokenizer we employ (Section 5).

Some compute-bound algorithms are inherently control-dominated and their data-flow reformulation is even more inefficient. Sometimes a more efficient, data-parallel algorithm exists that serves a similar purpose and can be used as a replacement (Class 3 in the figure), possibly with relaxed constraints. For example, in this work we replaced a traditional, exact document inversion algorithm with a faster, hash-based one that is subject to false positives (see Section 6). When an efficient replacement can not be found (Class 4), performance is generally low due to frequent branch misses and dependency stalls, no opportunities for SIMDization, and low ILP.

Control-dominated memory-bound algorithms can be optimized by pipelining accesses to main memory (Class 5), especially if the memory access patterns can be predicted in advance. This is the case with large, automaton-based multi-pattern string matching [15]. If such pipelining is not possible (Class 6), performance is even worse than Class 4 because round-trip latency to main memory can be very long (~830 clock cycles).

## 4. OVERALL DESIGN

The organization in Figure 1 does not map well onto the SPEs because stages operate on a single document at a time. The data struc-



**Figure 3: The four stages that compose our indexing design.**

tures required to process a document can be much larger than the LS. Rather, we employ the organization in Figure 3, where Stages 1–3 operate entirely on small blocks that fit into the LS. One run of Stages 1–3 loads an input text block from main memory to LS with a DMA operation, consumes that block, produces one block of index, and writes that block from LS to main memory. Stage 1 performs tokenization, Stage 2 performs inversion and Stage 3 sorts the output index block. Stages 1–3 form a pipeline as in Figure 9. Thanks to double-buffering, the time needed for DMA transfers to load input blocks from main memory and commit output blocks to main memory is hidden.

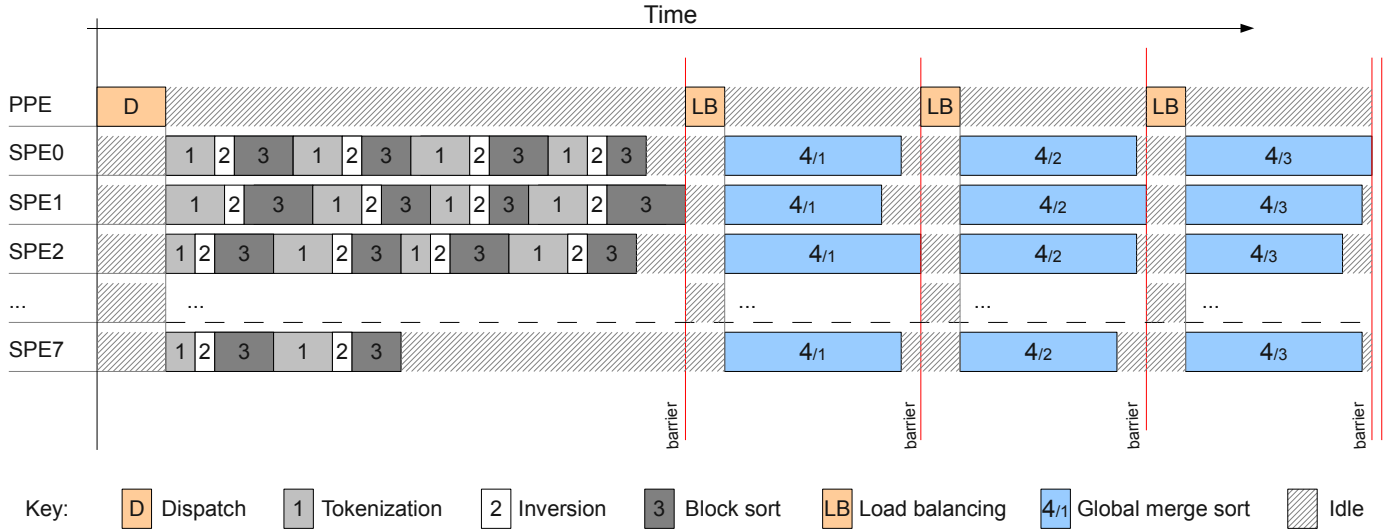
Finally, Stage 4 merges all the sorted blocks produced in Stage 3 to form a single, final, global index.

Stages 1–3 run sequentially on each of the SPEs, but each SPE works on a distinct partition of the input documents. Partitioning is done on the PPE at startup (the Dispatch task in Figure 4.). We assume that the collection fits entirely in main memory together with the output index<sup>2</sup>.

Table 1 reports the experimental results obtained by our indexer from the Kiwix [16] Wikipedia collection. Values refer to the PowerXCell 8i 3.2 GHz as available on the IBM QS22 blades. One Cell chip running our solution can index text at a sustained throughput of 161 Mbyte/s, which corresponds to 2,754 average Wikipedia articles per second. The majority of the time (68%) is required by the global merge stage. Stage 1 and 2 adapt very well to the architecture, showing a good value of CPI and negligible stall rates. Stage 3 and 4, on the other hand, adapt to a lesser degree to the Cell hardware, showing higher CPIs and much higher stall rates. This is indicative of the fact that branch removal in these stages was only partially successful.

Our solution shows a 4× performance advantage (Table 3) when compared to CLucene (a reference implementation of Lucene written in C++ [17]) running on a recent commodity machine like the quad-core Intel Q6600 processing identical input data [16]. Our Q6600 comparison figures result from four threads (one per core), each running an unmodified instance of CLucene version 0.9.21. For fairness, one must be aware that the unmodified CLucene source code does not contain explicit constructs designed to exploit SIMD (e.g., SSE instructions).

<sup>2</sup>Also, we ignore I/O issues related to transfers from/to permanent storage. Provided that an I/O subsystem with sufficient bandwidth is employed, a larger-scale design can guarantee that the indexer operates at sustained peak throughput by double-buffering input and output. In such a design, input data buffers and one output buffer are available to the indexer at all times.



**Figure 4: Our parallelization strategy and task organization. Stage 4 may comprise multiple passes (e.g., 4/1, 4/2, 4/3) executed in lockstep across the SPEs.**

	1. Tokenization	2. Doc. Inversion	3. Block Sort	4. Global Merge	End-to-End
<b>Latency:</b>					
Cycles per token / SPE	244.35	25.02	156.45	916.96	<b>1,342.79</b>
Cycles per token / Cell chip	30.54	3.13	19.56	114.62	<b>167.85</b>
Relative contribution	18.20%	1.86%	11.68%	68.29%	<b>100.00%</b>
<b>Throughput:</b>					
Consumed input Mbyte/s (1 Cell)	887.06	8,662.59	1385.40	236.38	<b>161.40</b>
Millions of tokens* per second (1 Cell)	101.49	991.14	158.51	27.05	<b>18.47</b>
Avg. Wikipedia articles* per second (1 Cell)	15,134.91	147,802.05	23,637.77	4,033.16	<b>2,754.16</b>
<b>Quality of mapping to the Cell platform:</b>					
CPI** (clocks per instruction)	0.75	0.75	1.75	1.80	
Branch stalls (% of clock cycles)	0.0%	0.2%	36.4%	22.7%	
Dependency stalls (% of clock cycles)	7.4%	2.3%	26.4%	33.1%	

The cost of tasks D and LB is not reported because they are negligible (0.042% and 0.014% of the entire workload, respectively).

\* Figures assume Wikipedia-like inputs: average token length = 8.74 bytes; average article = 6705.8 tokens, measurements by the authors.

\*\* The ideal CPI is 0.5 because SPEs have two pipelines and can issue up to two instructions per clock cycles. Higher values denote less efficient code.

**Table 1: Performance characterization of our Cell-based text indexer on Wikipedia-like data.**

In the scenario we consider, dispatching documents (D in Figure 4) in a balanced way is trivial because the time consumed by Stages 1–3 grows linearly with the size of the documents, which is known in advance. Moreover, in our scenario there are many small documents that allow fine-grained balancing. We adopt a rudimentary algorithm that assigns the next document to the SPE with the least load. This operates in linear time, provides a good load balance, and takes only 0.042% of the execution time.

When all SPEs complete Stages 1–3, a large number of small, sorted index blocks are available in main memory. Stage 4, in a parallel manner, merges these sorted blocks (potentially millions of them) into a single sorted index. Stage 4 may require multiple passes, (Figure 4 shows three) and their load balancing is assisted by a PPE task (LB in the figure). For one or more passes, the LB task employs a radix-histogram load balancing algorithm that we discuss in Section 8. This algorithm assigns to each SPE an independent and approximately equal fraction of the token space. Its cost is negligible (0.014% of the entire workload).

The next sections examine the algorithms and the performance of each stage in detail.

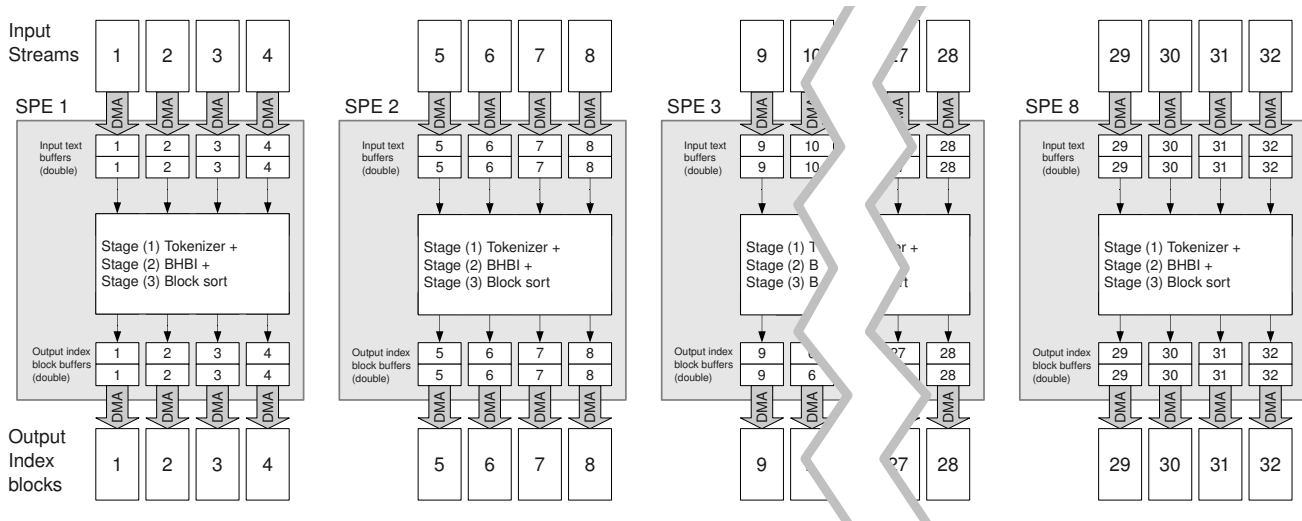
	Clock rate	Cores	Throughput	Advantage
IBM Cell	3.2 GHz	9	167.85	=1.0×
Intel P4 Northwood	3.2 GHz	1	11.82	14.20×
Intel Core 2 Quad Q6600	2.4 GHz	4	42.87	3.92×

**Table 3: Our solution has a significant performance advantage over a reference implementation of unmodified CLucene 0.9.21 [17] running on commodity Intel processors.**

## 5. TOKENIZATION

Tokenization divides the input text into tokens. This is an application of regular expression (regex) scanning. Regex scanners are sometimes written by hand, but more often they are Finite-State Machines (FSM) generated with automated tools. The most popular scanner generator is Vern Paxson’s flex [18].

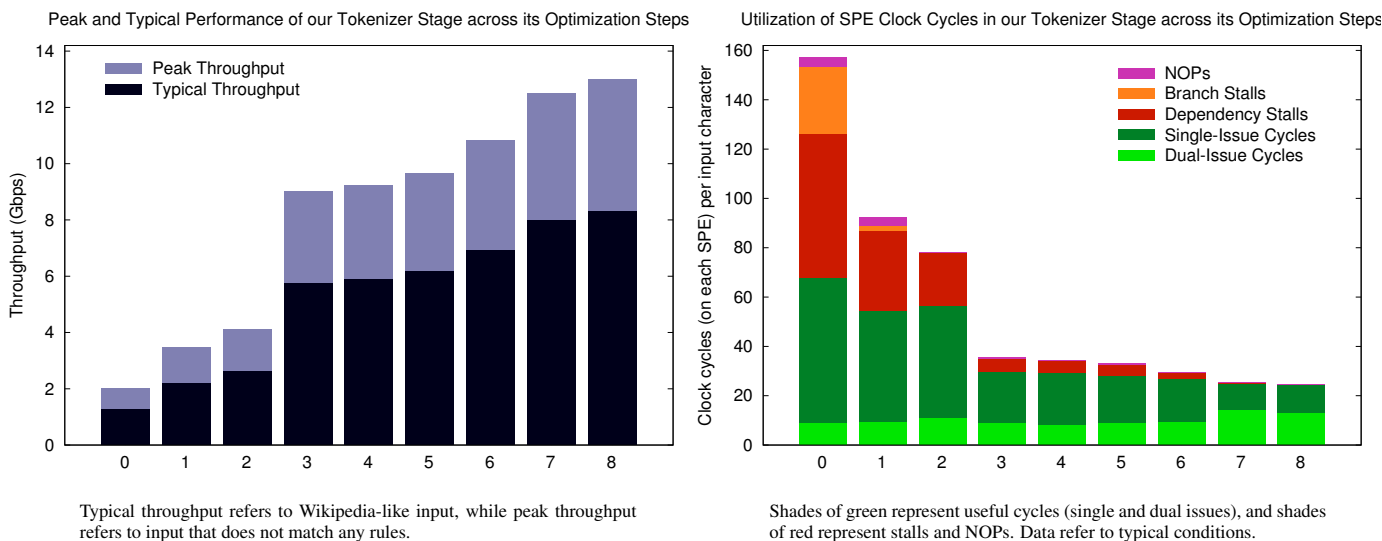
Unfortunately, the code these tools generate (including flex) has far-from-optimal performance on modern multi-core processors. No tools generate SIMD or multi-threaded code. Additionally, this code is very branch-intensive in nature, and this exacerbates the effect of



**Figure 5:** To exploit SIMD instructions, Stage 1 operate on four documents concurrently on each SPE. For this reasons, a run of Stages 1–3 takes four input blocks and produces four output index blocks. Input and output blocks are transferred from/to main memory via DMA transfers.

Optimization Step	Typical Throughput Gbyte/s	Cycles/char (on each SPE)	CPI	Instr's per char	NOPs	Branch Stall Rate	Dep. Stall Rate	Single Issue Rate	Dual Issue Rate	Used Regs
0 Vanilla flex x8 SPEs	166.6	157.31	1.76	89.58	2.48%	17.38%	36.97%	37.60%	5.57%	76
1 STT optimizations w/ ptr arithm	284.1	92.27	1.23	75.17	3.54%	2.52%	34.80%	48.80%	10.35%	56
2 Branchless	335.8	78.06	1.04	75.03	0.02%	0.01%	27.92%	57.99%	14.06%	68
3 4 Engines, SIMD	739.0	35.47	0.86	41.33	1.11%	0.01%	15.36%	58.24%	25.28%	89
4 4 Engines, SIMD, Unroll x4	757.7	34.60	0.87	39.97	0.86%	0.01%	14.62%	60.84%	23.67%	86
5 4 Engines, SIMD, Ux2, SW/pipe	791.1	33.14	0.81	40.74	1.19%	0.01%	14.09%	57.06%	27.65%	89
<b>6 4 Engines, SIMD, Ux16, SW/pipe</b>	<b>887.5</b>	<b>29.38</b>	<b>0.75</b>	<b>39.37</b>	<b>0.54%</b>	<b>0.01%</b>	<b>7.42%</b>	<b>59.62%</b>	<b>32.40%</b>	<b>89</b>
7 8 Engines, SIMD, Ux2, SW/pipe	1026.5	25.54	0.60	42.56	0.02%	0.02%	2.07%	41.93%	55.97%	72
8 8 Engines, SIMD, Ux16, SW/pipe	1066.8	24.57	0.61	40.52	0.02%	0.02%	1.05%	45.09%	53.83%	72
Unoptimized Reference	Throughput	(1 PPE)								
Original flex on 1 PPE	15.4	207.79								

**Table 2:** Performance of the optimization steps applied to the tokenizer. Typical throughput is per chip (8 SPEs), obtained running Lucene’s tokenizing rule set on Wikipedia pages. Optimization Step 6 (bold) is the one we employ in our solution.



Typical throughput refers to Wikipedia-like input, while peak throughput refers to input that does not match any rules.

Shades of green represent useful cycles (single and dual issues), and shades of red represent stalls and NOPs. Data refer to typical conditions.

**Figure 6:** As more optimizations are applied, performance grows (left) because the tokenizer needs fewer clock cycles (right) to process each input character.

branch misses on the Cell.

A recent work [19] has presented a technique to bridge this gap. The authors use flex to generate a first state-transition table (STT) from the user rules. Then, they replace the flex kernel with a Cell-optimized kernel they provide which operates on a re-encoded STT. The Cell-optimized kernel functions without branches and uses SIMD instructions that process the workload of 4 FSMs concurrently. We obtained the tools from the authors and we reuse them as Stage 1 in our text indexer.

The unmodified flex, when run on a Cell chip, would utilize only the PPE. This configuration runs the tokenizing ruleset of the Lucene with a throughput of 15.4 Mbyte/s (that corresponds to 208 PPE clock cycles spent to process a single token, assuming a Wikipedia-like input), as in the line “Unoptimized Reference” at the bottom of Table 2. The Cell-optimized kernel runs 67.7× faster. The speedup is due to algorithmic improvements that (1) use fewer instructions per FSM transition and (2) make the instruction schedule denser (decreasing the clocks per instructions (CPI) from 1.76 to 0.61).

Table 2 shows the impact of the optimizations on performance. Step 0 represents an unmodified flex running on each of the 8 SPEs on a chip: this yields an 10.6× speedup with respect to the PPE (i.e., each SPEs is 1.32× faster than the PPE). Step 1 uses a more efficient STT and faster pointer arithmetics, yielding a 1.7× speedup. Step 2 replaces the remaining branches with speculated code; the speedup (1.18×) is moderate, but it enables SIMD-ization. The removal of branches has the effect of eliminating nearly all branch miss stalls; see Steps 0–2 in Figure 6 (right).

Step 3 exploits SIMD instructions to process data corresponding to four independent FSMs, each operating on a distinct document. This requires a streaming double-buffered organization as in Figure 5. Each SPE runs four FSMs; thus the entire Cell chip maintains 32 documents streaming in and out at any given time. Performance grows to 739 Mbyte/s/chip, a 2.2× speedup over the best non-vectorized code.

The best SIMD code with four automata, after loop unrolling, software pipelining, manually reordered loads/stores, is Step 6 (bold face type in the table), and achieves a throughput of 887.51 Mbyte/s, which is 21.9× faster than flex on the PPE.

The authors propose further optimizations in Steps 7 and 8, that we did not implement. These further steps would double the FSMs from four to eight, providing an additional group of independent SIMD instructions to the compiler that it could use to fill dependency stalls and achieve better dual issues rates (ILP). In fact, dependency stalls almost disappear in Steps 7–8 in Figure 6 (right), increasing the performance by a factor of 1.2×. The disadvantage of this approach is that the number of input streams double from 32 to 64. Our LS memory budget is so tight that we can not afford this doubling.

## 6. DOCUMENT INVERSION

Traditional document inversion techniques do not map efficiently to an in-core stage on the Cell platform because of (1) the unbounded size of their working set, (2) their intense control flow, and (3) the inefficiency of their dynamic allocation and reallocation of data structures. We propose and employ a Blocked, Hash-Based Inversion technique (BHBI) designed to counter these issues. This section discusses its pros and cons and characterizes its performance.

Inversion algorithms customarily have been aware of the large latency difference between main memory and disk accesses. Blocked Sort-Based Indexing (BSBI, see Algorithm 1), a traditional document inversion algorithm [20], inverts fixed-size blocks of input at a time and commits them to permanent storage. This allows indexing documents arbitrarily larger than the available main memory.

On multi-core processors, the latency difference between core-local memories and main memory is also important. For example, on the

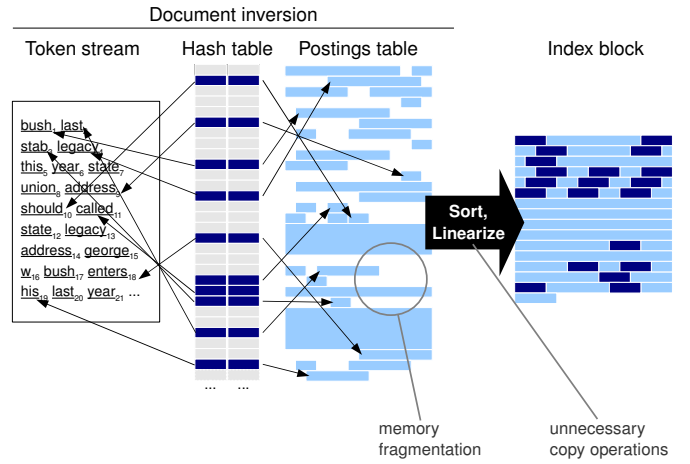


Figure 7: The traditional structure of a document inversion stage.

Cell’s SPEs, a load from the LS costs 6 clock cycles, but a load from main memory costs up to ~830. On the POWER6 [21], loads cost 2 cycles from the L1 cache, 20–26 cycles from the L2 cache (core-private), and ~450 from main memory. On the Intel Itanium, loads cost 2, 6, 22 and ~200 cycles from L1, L2 and L3 caches, and main memory, respectively. These large differences encourage the design of algorithms that can keep their entire working set in core-local memories.

Unfortunately, BSBI can not be mapped well to multi-core processors because its working set includes the term-termID dictionary<sup>3</sup> of the entire collection. This dictionary is unsuitable to fit in core-local memories because it may reach fractions of a Gigabytes for collections of interesting size.

**Algorithm 1** Blocked sort-based indexing (BSBI) [20]. The algorithm stores inverted blocks in files  $f_1, f_2, \dots, f_n$  and the merged index in  $f_{merged}$ .

```

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)

```

Single-Pass In-Memory Inversion (SPIMI) [22], described in Algorithm 2, overcomes these limitations by maintaining in its dictionary only the terms that appear in the current input block.

SPIMI can be adapted to run on an SPE, but with significant performance limitations. The fact that it allocates and reallocates postings lists dynamically (lines 7 and 12 of the algorithm) causes fragmented data structures (in the middle of Figure 7) and wastes scarce core-local memory. SPIMI’s hash table can be implemented with *probing* or *chaining*, but both have expensive or unpredictable access latencies. Also, postings tables must be defragmented into a linear data structures before being committed to main memory. All these operations are computationally expensive and require a complex control flow that prevents SIMDization and incurs strong branch penalties on the Cell.

<sup>3</sup>a mapping between each distinct term in the documents and a unique identifier assigned to it the first time the term is encountered.

---

**Algorithm 2** Inversion of one block in Single-Pass In-Memory Indexing (SPIMI) [22].

---

```

1: while  $\neg \text{empty}(\text{token\_stream})$  do
2:    $\text{output\_file} \leftarrow \text{NewFile}()$ 
3:    $\text{dictionary} \leftarrow \text{NewHashTable}()$ 
4:   while free memory available do
5:      $\text{token} \leftarrow \text{GetNextToken}(\text{token\_stream})$ 
6:     if  $\text{token} \in \text{dictionary}$  then
7:        $\text{postings\_list} \leftarrow \text{AddToDict}(\text{dictionary}, \text{token})$ 
8:     else
9:        $\text{postings\_list} \leftarrow \text{GetPostingsList}(\text{dictionary}, \text{token})$ 
10:    end if
11:    if  $\text{full}(\text{postings\_list})$  then
12:       $\text{postings\_list} \leftarrow \text{DoublePostingsList}(\text{dictionary}, \text{token})$ 
13:    end if
14:     $\text{AddToPostingsList}(\text{postings\_list}, \text{docID}(\text{token}))$ 
15:  end while
16:   $\text{sorted\_terms} \leftarrow \text{SortTerms}(\text{dictionary})$ 
17:   $\text{WriteBlockToDisk}(\text{sorted\_terms}, \text{dictionary}, \text{output\_file})$ 
18: end while

```

---

Blocked Hash-Based Inversion (BHBI), the algorithm we propose (Algorithm 3) overcomes these shortcomings by dispensing with need for a term dictionary. BHBI creates a hash of each input token  $t$  that is used as a term identifier, and adds a line  $(\text{hash}(t), \text{docID}, i_t)$  to the token-hash table, as in Figure 9. Note that the token-hash table is not an ordered hash table; entries are added sequentially after each tokenization. Input/output buffers are pre-dimensioned to use the local memories efficiently, with no need for dynamic allocation.

---

**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:	$\text{docID}$ , 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;

Symbol  $\leftarrow$  denotes a variable assignment in local memory.  
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:      $xb_i \leftarrow (\text{hash}(tb_i), \text{docID}, w + i)$ 
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

```

---

BHBI uses a storage scheme called *single payload* (in the terminology of Melnik et al. [23]) 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  $(\text{hash}(t), \text{docID}, i_1), (\text{hash}(t), \text{docID}, i_2), \dots$

BHBI uses no term dictionary. It constructs a hash of each token, which it uses as its term identifier. To avoid confusion, note that the hashes themselves are stored and sorted; they are not used as an index into 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 following advantages. There is no dynamic memory allocation and reallocation, and no consequent unnecessary copy operations. Explicit buffer management allows fitting the working set into small core-local memories, and employing double buffering. Each iteration of the inversion kernel loop (lines 4–6 in Algorithm 3) processes independent inputs and produces independent outputs, and it is free of loop-carried dependencies and is SIMDizable/unrollable *ad libitum*. There is no hash table, and no time consuming probing.

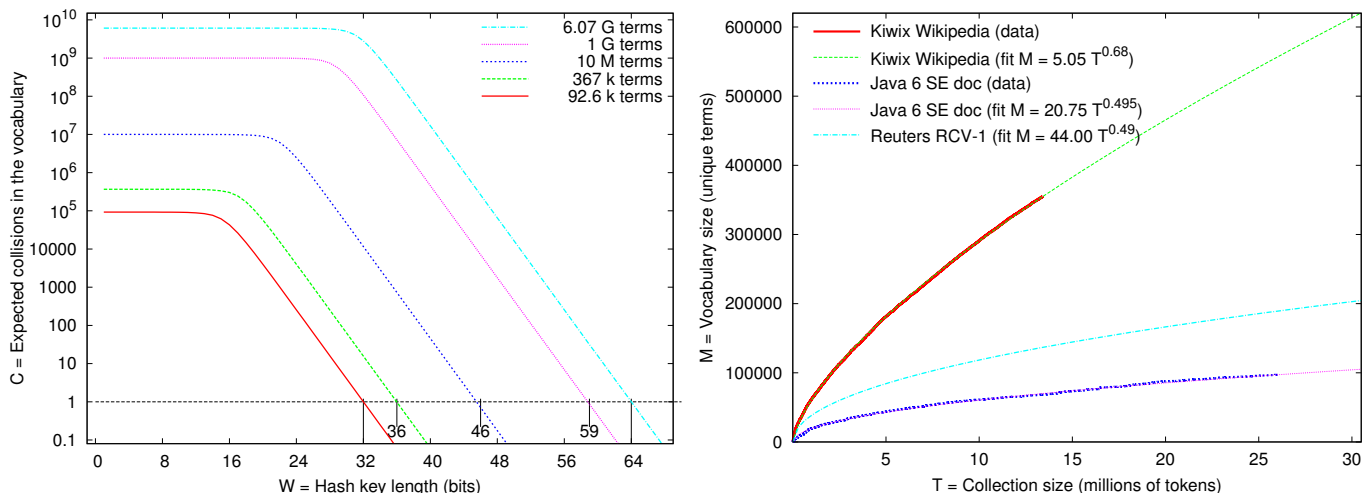
Expensive variable-length string manipulations are replaced with inexpensive hashing, which also speeds up sorting, since comparing aligned entries costs much less than comparing variable length strings, as Inoue et al. [24] show.

BHBI also allows three additional optimizations, whose impacts on performance depend on the target platform: (1) 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; (2) by matching the size of input and output entries, BSBI can operate *in place* saving a significant amount of memory; (3) a later sort step can assume that entries with the same hash are already in the right order by construction; i.e., entry  $(h, \text{docID}, w + 1)$  necessarily appears after  $(h, \text{docID}, w)$ . Our experimental implementation takes advantage of all of these.

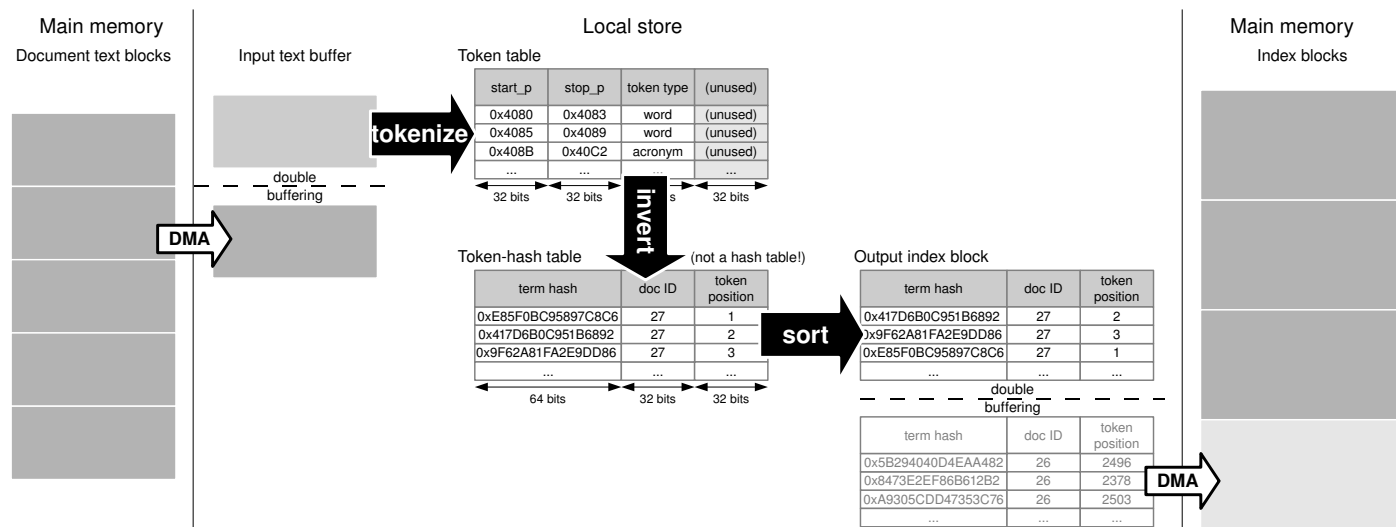
Stage 2 as just described can be optimized very well on the SPE, especially because entries are aligned and independent of each other. We use a SIMD hash function, SIMD string manipulation, and loop unrolling, achieving a  $28.91\times$  speedup with respect to a naïve implementation based on C library string functions and a rudimentary DJB2 hash implementation, as Figure 11(a) shows. We also achieved a very high resource utilization, e.g., our CPI is 0.75, as in Figure 11(c). This is close to 0.5, the ideal CPI for the SPE.

BHBI’s main limitations are: (1) the index no longer contains plain-text terms, and (2) hashes can collide. The first is acceptable, because the index contains references to the tokens in the original document. The second 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 [13]. However, analogous to the statistics of the *common birthdays problem*, the expected number of collisions (common birthdays)  $\tilde{C}$  incurred by a *good*  $W$ -bit wide hashing function over a collection that has  $M$  distinct terms behaves as in Figure 8 (left). If we assume that  $\tilde{C} < 1$  is safe, (i.e., it has an acceptably small probability of collisions) a 32-bit hash is safe up to 92,682 terms. The Kiwix Wikipedia dump [16] (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 [25] ( $M = k \cdot T^b$ ), plotted in Figure 8 (right) 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



**Figure 8:** These two charts allow computing the maximum collection size that can be safely indexed with a hash of a given bit-width: the relationship between a hash width and the expected collisions over a dictionary (left), and Heaps' Law (right) that correlates the size of a collection  $T$  with the size of its vocabulary  $M$ . Examples include the Kiwix Wikipedia [16], Java 6 documentation, Reuters' RCV1 [20].



**Figure 9:** Data flow of Stages 1–3 of our indexer for the Cell processor.

Wikipedia-like collection safely indexed with a 64-bit hash according to Heaps' Law would be  $4.83 \cdot 10^{13}$  tokens = 422.42 Terabytes = 7.2 billion average Wikipedia pages.

## 7. BLOCK SORT

The design of efficient in-core sort algorithms for the Cell is an open problem. In fact, asymptotic analysis is of little help here because small amounts of data are processed at a time, due to the paucity of the LS, and branch penalties tax algorithms with a complex control flow.

We employ a block sort implementation that runs, on average, twice as fast as quicksort for the short arrays, as Figures 10(a,b) show. It uses a cycle of the *American flag* sort [26] followed by a bubble sort.

The American flag sort is a variant of a linear-time histogram sort that initially distributes items into hundreds of buckets. The first step counts the number of items in each bucket and the second step com-

putes where each bucket will start in a reconstituted array. The last step moves items to their proper bucket in the reconstituted array. We use only one cycle of the sort with 256 buckets, the number of buckets being based on using the leftmost eight bits of the key (the hash) as the bucket selector. Then, we use an unrolled, SIMDized *bubble sort* on each of the bucket. Despite its high asymptotic cost ( $O(N^2)$ ), bubble sort runs fast on the small numbers of per-bucket keys, and it is amenable to unrolling and SIMDization. Results are promising, but the algorithm is still control-intensive and suffers from branch miss stalls.

## 8. GLOBAL MERGE

This section describes Stage 4 of our indexer. We call any sorted sequence of keys (the hashes) a *run*. Stage 4 is a multi-way merge of runs in which all the index blocks produced during tokenization, inversion and block sort are merged into a single sorted index that

Optimization	CPI	Cycles/ token	Mtoken/s (1 SPE)	Mbyte/s (1 SPE)	Speedup
1 Mergesort	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 10(a): The block sort implementation we optimized for the purpose of this work achieves a  $\sim 2\times$  speedup with respect to quicksort. Still, resource utilization is improvable, as indicated by the relatively high CPI.

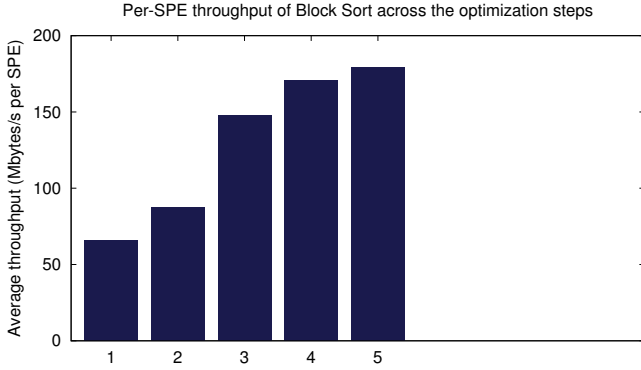


Figure 10(b): Average throughput per SPE, in Mbyte/s of input text processed.

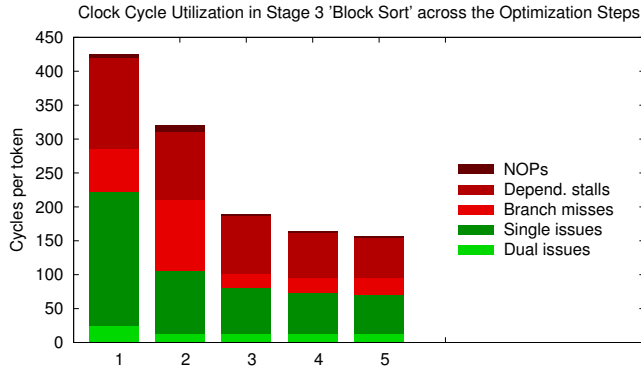


Figure 10(c): Utilization of the clock cycles across the optimization steps. Shades of green indicate productive cycles (single or dual issues); shades of red indicate unproductive ones. We achieved a significant overall speedup, but not a significant increase in resource utilization.

**Figure 10: (a),(b),(c): Performance of the block sort stage across the optimization steps.**

contains the entries of the entire collection. In this stage, input and output runs might together occupy nearly the entire amount of available main memory. For the sake of simplicity, our implementation does not consider the extension of this merge to disk storage.

Unlike Steps 1–3, Step 4 is *out-of-core*. The runs are in main memory and exhibit relatively little data locality, so they can not be streamed easily through the LS.

In our tests, the algorithm that best maps the problem to the Cell architecture is a tournament-tree merge. We assume that the reader is familiar with the basics of a tournament-tree merge [27]. A tournament tree has as many leaves and as many vertexes as there are input runs. Each pair of leaves has one level-1 descendant vertex, and each pair of level-1 descendants have one level-2 descendant, continuing on until reaching the final descendant, which is the root vertex (for simplicity we assume that the number of input runs is a power of 2).

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

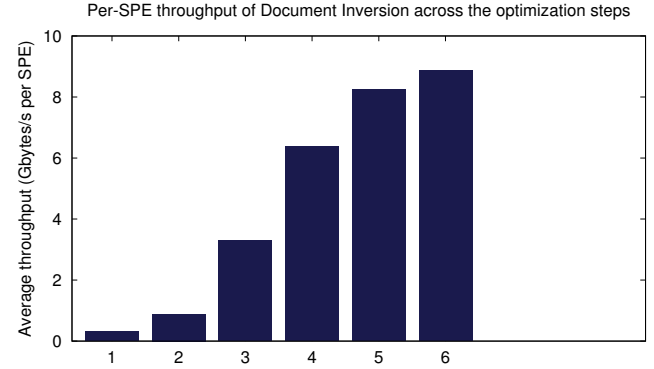


Figure 11(b): Average throughput per SPE, in Gbyte/s of processed input text.

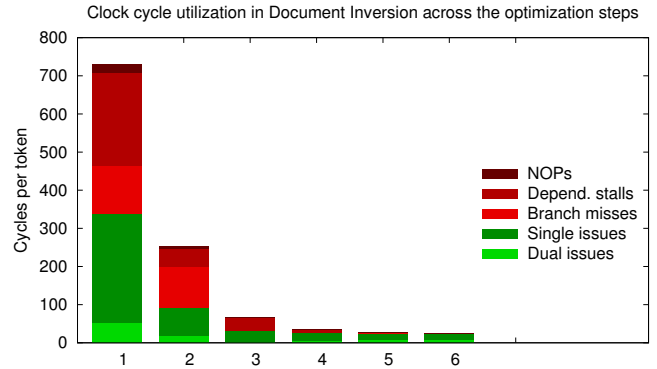
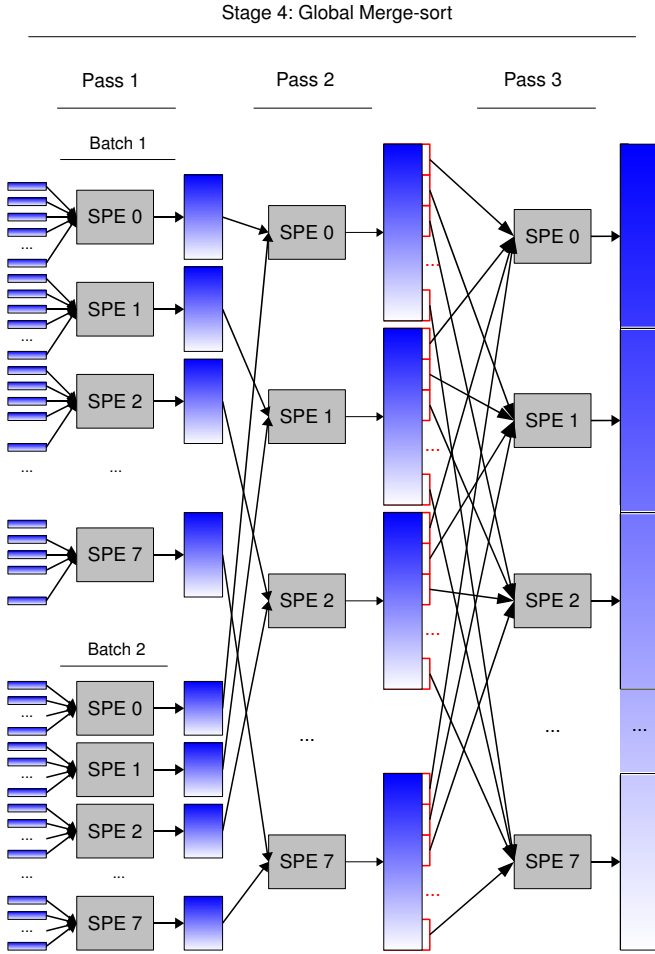


Figure 11(c): Utilization of the clock cycles across the optimization steps. Shades of green indicate productive cycles (single or dual issues); shades of red indicate unproductive ones. The chart shows that our optimizations were able to get rid of the branch misses, a significant portion of dependency misses, and achieve a good rate of dual issues.

**Figure 11: (a),(b),(c): Performance of document inversion across the optimization steps.**

The tournament-tree is initialized by placing a smaller-than-the-smallest key in each vertex. Note that each key in each run that is to be merged must be tagged with the unique identifier of the run from which it originates. Initially, each leaf, in sequence, takes the first key from its corresponding input run, and that key competes in the tournament until the smallest key appears in the root vertex. Initialization is complete when any key reaching the root vertex is larger than smaller-than-the-smallest key; that key is the smallest key from all of the runs being merged. All of the smaller-than-the-smallest keys are discarded; the number discarded will be equal to the number of leaves in the tree.

The key in the root vertex is moved into the first sequential key of the output run. The unique run tag of the removed key determines which run is the next to supply a key to the tournament, and the next key from that designated run, entering through its corresponding leaf, is used to begin the next cycle of tournament competition. The winning (smallest) key of each tournament cycle is moved into the next sequential key location in the output run, its unique run identifier is



**Figure 12: Our global merge stage adopts a parallelization approach based on three passes. In Pass 1 and 2, each SPE accepts multiple, entire runs as inputs and produces one output run. In Pass 3, each of the  $N$  SPEs takes as an input a fraction ( $\approx 1/N$ ) from each of the  $N$  runs produced in Pass 2, and produces a fraction of the single, final, output run.**

extracted, and the process is repeated.

The number of compares required to determine the next key in the merged run is  $\lceil \log_2(R) \rceil$ , where  $R$  is the number of runs being merged. Time complexity is therefore  $K \lceil \log_2(R) \rceil$ , where  $K$  is the total number of keys in the final run. When all the keys from a particular run have been propagated through the tournament tree, a special infinity (larger-than-the-largest) key is used to signify the end of that run. The first infinity key to reach the root vertex signifies merge completion.

In most indexes of interest, long sequences of keys having the same hash regularly occur in most runs. If a single bit in each key is used to signify that the hash of the next key is the same as the hash of the present key, the next key can be declared a tournament winner immediately and placed into the next sequential key location in the output run, thereby bypassing all comparisons required in the tournament tree. This leads to a substantial improvement in performance, and is included in the performance measurements reported.

Our implementation employs a double buffer between input runs and leafs to mitigate the latency of DMA reads from main memory, and also between the root vertex and the output run for the same rea-

son. Each vertex contains one 128-bit key, representing one token from the input document collection, as described in the previous sections. Our parallelization strategy is Figure 12. The first passes (1 and 2 in the figure) reduce the number of total runs until they are  $N$ , which is the number of processing elements, eight in our case. Load balancing is trivial in these early passes.

The last pass requires splitting the merge of  $N$  runs across  $N$  processor in a balanced way, which is less trivial. To do so, we employ the *radix histogram* of each run and of the ensemble of runs. The histogram bins tell how many entries in each run begin with a given prefix. We use 8-bit prefixes that correspond to histograms having 256 bins. The counts in the bins of the ensemble histogram are the sums of the counts in corresponding bins of the  $N$  individual histograms. We use the ensemble histogram to divide the radix domain (0,...,255) into  $N$  partitions of contiguous bins, on radix boundaries, such that the partitions have roughly equal count sums. This division tends to minimize the load imbalance among the  $N$  SPU's. The individual runs are then divided on the same radix boundaries, based on their individual histograms. In this approach, values in each partition do not overlap or interact with adjacent ones, and therefore each can be merged independently. Additionally, memory coordinates of each partition can be determined exactly a priori, in both the individual runs and the ensemble run, by summing values in their respective histogram bins.

## 9. RELATED WORK

Since a multi-core is similar to a distributed system on a chip, designers attempted to reuse distributed-systems programming models on multi-cores. For example, MapReduce has been ported on multi-cores [28] like the Cell [29] and nVidia's GPGPUs [30]. Multi-cores, though, differ so much from distributed systems that significant performance is left unexploited. For example, MapReduce makes no attempt at exposing DLP to the hardware.

Despite the need for indexing, little work has addressed the mapping of indexing tasks onto multi-cores, except for text matching [31, 32, 33], or sorting [24, 34]. Authors have focused on in-main-memory index merging strategies [35]; or on index re-ordering techniques [36] that reduce query latency, or compressed text databases [37] that reduce space requirements, all at the expenses of indexing speed.

No work addresses indexing with reference to multi-cores and data-level parallelism. We address this gap, focusing on algorithmic choices and implementation details that allow high throughput with a basic set of indexing features.

## 10. CONCLUSIONS

We have examined text indexing, an emerging workload, on the Cell processor, an emerging multi-core architecture.

Our experience suggests that the traditional algorithms employed in the tasks of text indexing are often incapable of exploiting the amount and degrees of parallelism made available by a peculiar data-flow oriented machine like the Cell processor.

We propose a text indexing flow that features alternative algorithms explicitly designed to address these shortcomings, and we provide an analysis of their performance. We find that two stages of the flow (tokenization and document inversion) adapt extremely well to the architecture, delivering impressive throughput. The remaining stages (block sort and global merge) exhibit a residual amount of control-flow that hinders a good mapping to the Cell, causing branch misses and dependency stalls that are difficult to eliminate.

Nevertheless, the overall end-to-end performance delivered by our indexer compares favorably against a reference indexer implementation running on a commodity multi-core machine. Our solution, in a per-chip comparison, is the fastest text indexer that we are aware of.

## 11. REFERENCES

- [1] IDC Corporation. The expanding digital universe. *White Paper*, March 2007.
- [2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, Dec. 2004.
- [3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Mauerer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, pages 589–604, July/September 2005.
- [4] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency, Mar. 2008.
- [5] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerma, 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.
- [6] H. P. Hofstee. Efficient Processor Architecture and the Cell Processor. In *Conference on High Performance Computing Architectures (HPCA'05)*, February 2005.
- [7] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3), May/June 2006.
- [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] J. L. Hennessy, D. A. Patterson, K. Asanovic, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. *Computer architecture: a quantitative approach, fourth edition*. Morgan Kaufmann, 2006.
- [10] The Green500 List, <http://www.green500.org/lists/2008/11/list.php>, Nov. 2008.
- [11] IBM. Roadrunner smashes the Petaflop barrier, press release, June 2008.
- [12] Top 500 supercomputers, <http://www.top500.org/>, Nov. 2008.
- [13] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.
- [14] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
- [15] D. P. Scarpazza, O. Villa, and F. Petrini. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor. In *22nd IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, Florida, Apr. 2008.
- [16] Kiwix, version 0.5, [www.kiwix.org](http://www.kiwix.org), Mar. 2007.
- [17] The Apache Software Foundation. CLucene – a C++ search engine, <http://sourceforge.net/projects/clucene/>.
- [18] V. Paxson. flex – a fast lexical analyzer generator, 1988.
- [19] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In *23rd Intl. Conference on Supercomputing (ICS'09)*, Yorktown Heights, New York, USA, June 2009.
- [20] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008.
- [21] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr. Design and implementation of the POWER6 microprocessor. *IEEE Journal on Solid-State Circuits*, 43:21–28, Jan. 2008.
- [22] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54:713–729, 2003.
- [23] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *ACM Tenth Intl. World Wide Web Conference (WWW10)*, Hong Kong, May 2001.
- [24] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *16th Intl. Conference on Parallel Architecture and Compilers (PACT'07)*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.
- [25] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, Inc., Orlando, FL, USA, 1978.
- [26] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6:5–27, 1993.
- [27] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Ed.* Addison-Wesley, 1998.
- [28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *13th IEEE Intl. Symposium on High-Performance Computer Architecture (HPCA-13)*, pages 13–24, Phoenix, AZ, Feb. 2007. IEEE Computer Society.
- [29] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell B.E. architecture. Technical Report CS-TR-2007-1625, University of Wisconsin Computer Sciences, Oct. 2007.
- [30] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *17th Intl. Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, Toronto, Canada, Oct. 2008.
- [31] O. Villa, D. P. Scarpazza, and F. Petrini. Accelerating real-time string searching with multicore processors. *IEEE Computer*, 41:42–50, Apr. 2008.
- [32] R. D. Cameron. Method and apparatus for processing character streams, U.S. Patent 7400271, July 2008.
- [33] R. D. Cameron. A case study in SIMD text processing with parallel bit streams - UTF-8 to UTF-16 transcoding. In *2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 91–98, Salt Lake City, Utah, Feb. 2008.
- [34] B. Gedik, R. Bordawekar, and P. S. Yu. Cellsort: High performance sorting on the cell processor. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *33rd Intl. Conference on Very Large Data Bases (VLDB'07)*, pages 1286–1207. ACM, 2007.
- [35] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *14th ACM Conference on Information and Knowledge Management (CIKM'05)*, pages 317–318, 2005.
- [36] S. Garcia, H. E. Williams, and A. Cannane. Access-ordered indexes. In *New Topological Descriptors, Journal of Chemical Information and Computer Sciences*, pages 7–14. Zealand, 2004.
- [37] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2:2004, 2002.