Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms

Timothy Furtak, José Nelson Amaral, Robert Niewiadomski
{*furtak,amaral, niewiado*}*@cs.ualberta.ca*
Department of Computing Science,
University of Alberta, Edmonton, Canada

Abstract

Most contemporary processors offer some version of Single Instruction Multiple Data (SIMD) machinery — vector registers and instructions to manipulate data stored in such registers. The central idea of this paper is to use these SIMD resources to improve the performance of the tail of recursive algorithms. When a recursive computation reaches a set threshold, data is loaded into the vector registers, manipulated in-register, and the result stored back to memory. Four implementations of sorting with two different SIMD machinery — x86-64's SSE2 and G5's AltiVec — demonstrate that this idea delivers significant performance improvement. The improvements provided by the tail optimization of sorting are orthogonal to the gains obtained through empirical search for a suitable sorting algorithm [10]. When integrated with the Dynamically Tuned Sorting Library (DTSL), this new code generation strategy improves the performance of DTSL by up to 18%. Performance of d-heaps is similarly improved by up to 35%.

1 Introduction

This paper addresses the automatic generation of efficient code to sort short sequences of values. The idea is that an ahead-of-time optimizer searches for fast code for several sequence lengths and machine configurations. Then the compiler can simply instantiate such code when generating an optimized library. While algorithm-specific optimizations and empirical search have long been used both for scientific computation and for large parallel machines [4, 5, 16, 18], only recently these techniques were applied to integer-intensive, symbolic, computation. Li *et al.* developed the Dynamically Tuned Sorting Library that adapts to the characteristics of the input to be sorted [10]. The main contribution of this paper is the insight that the resources implemented in contemporary processors to enable SIMD computations can be put to good use to improve the performance of sorting short sequences. As demonstrated in this work the effective use of these SIMD resources improves performance through the reduction of memory references and increase in instruction level parallelism.

The initial inspiration for this work was the need for fast sorting of short sequences in the implementation of graphics rendering in interactive video-game applications. In such applications it is often necessary to decide, for each pixel of the image, what is the order of the elements that should be displayed [2]. Even though Z-buffer pixel-ordering computations are typically handled by a specialized Graphics Processing Unit (GPU), there are plenty of similar ordering computations that are done by the Central Processing Unit (CPU) in computer games. For instance, sorting is used to characterize the intensity of the various light sources that illuminate a character. Moreover, contemporary video-game application have at their disposal a rich supply of SIMD registers and instructions. For example, the PowerPC-based XBox 360 hardware features 128 AltiVec registers on each of its three cores along with an expanded set of AltiVec instructions. In addition to interactive video-game applications, sorting of short sequences is also present in particule-physics simulation applications.

Thus, using SIMD registers and instructions to sort small sequences is natural. Once a solution was created, applying it to the short sequences that must be sorted at the tail-end of standard recursive sorting algorithms was the next logical step. The experimental evaluation of the new vector-register-based sorting algorithms presented in this paper use commodity processors (x86-64 and G5) and extensions to the DTSL library because these machines and algorithms are more readily available and exploitable than proprietary video-game hardware and software. The algorithms presented are effective for sorting short sequences of floating-point or integer values (keys), and pairs comprised of a key and a memory address, *i.e.* key-pointer pairs, as well as computing the index of a specific (minimum or maximum) element.

Three new SIMD-based algorithms use the concept of sorting networks that are effective to sort small sets of numbers. Section 2 describes: (1) the operation of standard sorting networks; (2) how the SIMD vectors can be used to implement sorting networks; and (3) how a code generator can instantiate optimized vector code for sorting networks operating in sequences of any length. The main contributions in this paper are:

- three algorithms that use the SIMD machinery of contemporary processors for efficient in-register sorting of short sequences;

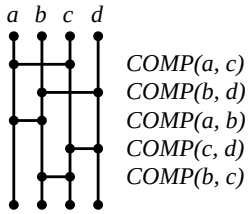

Figure 1: A 4-element sorting network.

- a method to use iterative-deepening search to find fast SIMD instruction sequences to move data within the SIMD registers;
- a method to integrate these algorithms in an optimized general-purpose sorting library;
- a method to compute the minimum element in an array, with applications to d-heaps;
- and an extensive experimental study in three different processors that demonstrate up to 18% improvement in the performance of DTSL and up to 35% in d-heaps. This study also indicates that the elimination of loads, stores, branches, and branch mispredictions correlates well with the improved performance.

Section 3 describes two algorithms that combine a first-pass sorting in the SIMD registers with a second-pass sorting in memory. Section 4 describes an algorithm that sorts shorter sequences completely within the SIMD registers, thus eliminating branch instructions altogether. Section 5 describes how to extend these key-sorting algorithms to sort key-pointer pairs. Section 6 uses similar techniques to speed up *heapify-down* operations in d-heaps. The experimental evaluation is presented in Section 7.

2 Sorting Networks

The inputs to an in-place *comparator*, $COMP(a, b)$, are two storage units — memory locations, registers, or vector-register elements — a and b, each containing a numerical input. After the comparator executes, the lower numerical value is stored in a and the higher numerical value is stored in b. Knuth describes a *comparator network* as a device that applies a fixed sequence of comparator operators to an input vector of a given size [7]. When a comparator network produces a sorted output for any possible input sequence, it is called a *sorting network*. The *size* of a sorting network is the total number of comparators in the network. The *depth* of a sorting network is the length of the critical path in its dependence graph. Therefore the depth provides a bound for the parallel execution of the sorting network, while the size provides a bound for a sequential execution.

An example of a sorting network with size 5 and depth 3 is shown in Fig. 1. The network is depicted as a set of value-carrying vertical rails and comparators. Values flow from top to bottom. A heavy dot at a line crossing indicates that the value at the vertical rail is an input to the comparator represented by the horizontal line. A comparator moves the larger value to the left, and the smaller value to the right. For instance, if the inputs are $a = 7$, $b = 2$,

$c = 5$, $d = 9$, then the sorted output at the bottom of the sorting network is $a = 9$, $b = 7$, $c = 5$, and $d = 2$. The value 9 moves from rail d to rail b at $COMP(b, d)$, and then moves from rail b to rail a at $COMP(a, b)$.

Although several algorithms are available to generate code for sorting networks, Batcher's "odd-even mergesort" algorithm is often chosen for its efficiency [1]. Batcher's algorithm uses $O(n \log^2 n)$ comparators and has a depth of $O(\log^2 n)$. Sorting networks can be efficiently implemented in processors that provide a *min* and a *max* instruction. Sorting networks implemented with these instructions avoid the performance penalties of branch miss-predictions incurred by traditional branch-based sorting implementations. The experimental results in Section 7 indicate that eliminating branches in the code of sorting networks is a significant win in contemporary processors.

2.1 Supporting Hardware

Consider a machine that has the following *min* and *max* instructions:

$$min(a, b) = \begin{cases} a : a \leq b \\ b : \text{otherwise} \end{cases}, \text{ and } max(a, b) = \begin{cases} a : a \geq b \\ b : \text{otherwise} \end{cases}.$$

The comparator required by a sorting network is easily constructed using these two operations, a copy instruction, and a temporary variable. For instance, such instructions are available in the x86-64 architectures supporting the SSE2 min and max operations that return the minimum (maximum) packed single-precision floating-point values [6].[1]

The extension of sorting networks to operate on vector instructions requires the definition of vectorized min and max instructions.[2] For input vectors A and B, $|A| = |B| = n$, let $C = min(A, B)$ be the element-wise minimum vector, such that $C_i = min(A_i, B_i), 1 \leq i \leq n$. The vectorized max instruction is defined similarly. The *width* of a (vectorized) sorting network refers to the number of vectors being sorted. Given an ordered list of vectors $X^1, X^2, \ldots, X^n$, a *stream* of data is formed by selecting the i^{th} element from each vector in order, thus the i^{th} stream is $X_i^1, X_i^2, \ldots, X_i^n$.

For instance, the x86-64 architecture has 16 XMM vector registers, and each register can hold 4 floating-point values. Therefore, sorting the values in n XMM registers using a sorting network produces 4 sorted streams of data of length n. Up to 15 XMM registers can be used, *i.e.* $1 \leq n < 16$, because one register must be reserved as temporary storage for the swap of values in the comparator.

This compare-and-swap machinery offers several advantages to sort a small set of values that fits within the SIMD registers: (1) its operation is unconditional and data-independent; (2) it is inherently branch-free, and thus free of branch-prediction performance penalties; (3) it increases the bandwidth of sorting by enabling the SIMD instruction-level parallelism; and (4) each compare-and-swap requires the execution of only 3 instructions.

A code generator must be able to generate code to sort sequences of any length in a machine with $n + 1$ SIMD registers. The solution is to define size-optimal sorting networks that use $1, 2, \ldots, n$ registers. The optimal code for the implementation of each of these sorting networks is pre-generated and stored in a small codebase available to the code generator

[1] SSE stands for Streaming SIMD Extensions. SSE2 improves upon the original SSE.

[2] These vector instructions are called a SIMD extension.

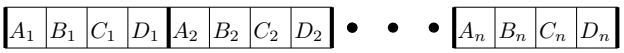

Figure 2: Interleaved sorted streams from n 4-element SIMD registers. The first register contains elements A_1, B_1, C_1, and D_1. $A_1 \leq A_2 \leq \cdots \leq A_n$, etc.

for deployment. Once data has been loaded into the SIMD registers the code generator instantiates the code to perform the comparator operations specified by the sorting network, and integrates the resulting streams.

3 Stream-Based Two-Pass Sorting

The first two SIMD-based sorting algorithms discussed in this paper operate in two phases. In the first phase the SIMD registers and instructions are used to generate a partially-sorted output. In the second phase a standard sorting algorithm — insertion sort and mergesort are investigated in this paper — finishes the sorting. The choice of algorithm for the second phase dictates the best data organization for the first one.

For the first phase, consider the use of the SIMD sorting machinery described in Section 2 for the task of sorting a sequence of $k * n$ values using n SIMD registers, each register capable of storing k values. Each group of k values is loaded from memory into a separate SIMD register. For a moment, assume that the start of the sequence is aligned for such a load operation. The sorting machinery is then applied to produce k sorted streams of length n, and the sorted streams are written back in-place to memory in an interleaved form. The organization of the data in memory for $k = 4$ is shown in Fig. 2. After sorting, $A_1 \leq A_2 \leq \ldots \leq A_n$, $B_1 \leq B_2 \leq \ldots \leq B_n$, etc.

After this initial sorting the ordering relationship between elements from separate streams, A_i, B_i, C_i, and D_i, is still unknown. Now the output from the vectorized sorting network must undergo an additional sorting pass. Let us examine the use of insertion sort and mergesort to finish sorting this partially sorted output.

3.1 Second Pass with Insertion Sort

A standard insertion-sort algorithm may be used to sort the output of the SIMD-based sorting network. Insertion sort delivers the best performance when its input is mostly sorted because the algorithm does not have to move elements very far. Thus a potential issue with using insertion sort as a second pass is how the data should be loaded into the SIMD vectors in the first phase to produce the most favorable input for insertion sort.

Consider an input sequence of S values, and a machine with $n+1$ SIMD vectors. Each vector can store up to k values. Let $m = \lceil S/k \rceil$. If $m \leq n$ the entire array can be loaded into the SIMD registers, sorted, and written back in-place. Then a call to insertion sort will finish sorting the entire sequence.

If $m > n$, an in-place algorithm divides the array into subsets small enough to fit in the vector registers, sorts them with a sorting network, and writes each sorted subset back to the same locations.

A naive approach would simply divide the array into $\lceil m/n \rceil$ almost equal-sized blocks. However, if the data is uniformly distributed this partition results in $\lceil m/n \rceil$ similar blocks, one after the other. The problem is that small elements from the last block would have similar values to the small elements from the first block, and would require insertion sort to move many elements to far positions to combine these blocks.

A better approach is to load the blocks into the SIMD registers in a strided fashion. Consider for example $n = 4$ and $m = 12$ which requires three sorting network calls. Instead of the first call acting on elements A^1, A^2, A^3, and A^4, it acts on A^1, A^4, A^7, and A^{10}. The second call acts on elements A^2, A^5, A^8, and A^{11}, and the third on A^3, A^6, A^9, and A^{12}. In this way the small values in the array are likely to end up in A^1, A^2, and A^3. A stride width greater than one improves insertion sort performance in cases of uniform or mostly-sorted distributions. In this paper, this strided version of the vectorized sorting network followed by an insertion sort pass is called ISort.

3.2 Second Pass with Mergesort

The mergesort algorithm, called MSort, uses a fixed-sized block of temporary storage T that is large enough to hold the entire array A. Because the SIMD-based sorting is applied to small sequences this array will not be large in practice. MSort proceeds as follows. Compute the number of blocks of data to be sorted, $\lceil m/n \rceil$, as width, and allocate temporary space T.

Call the sorting network on each block from A and store the sorted streams to T. The *Q-MERGE* algorithm described by Wickremesinghe *et al.* [17] is now used to store the sorted data into A: (1) Build a heap containing the first element in each stream, and associate with each element a pointer to the next element in its stream; (2) Repeatedly extract the minimum element from the heap. During the extraction, replace the removed element with the next element in its stream, and rebuild the heap.

With a small number of streams, sufficient registers may be available to contain the entire heap. Heapify operations are then efficient and the only flow of data to/from memory is to fetch the next item from a stream or to store the next value to A. For heaps that are too large to fit within the available registers, in-memory heap code may be used. Maintenance operations on small heaps may be written using the known register locations of elements, avoiding potentially costly memory accesses and pointer indirections.

MSort uses one merge heap, with the number of inputs being a multiple of v. That is, each heap completely handles the output from one or more vectorized sorting network calls. Further, only heaps which may be contained within the available registers are considered.

Further optimizations include placing a sentinel value of infinity at the end of each stream to avoid checking if streams are empty [17]. Once the sentinel is loaded into the head it will sink to the bottom. When any sentinel is extracted from the heap the sorting is complete.

Each sorting network call places elements from the same stream a constant distance away from each other. Thus the next element on a stream can be found by adding a constant offset to the address of the current element, which makes the maintenance of the "next element" pointer in the heap straightforward.

Table 1: SSE2 instructions used in the example of Fig. 3

Instruction	Description
`movaps Ra, Rb`	copy the contents of `Ra` to `Rb`
`shufps Ra, Rb, i`	copy 2 elements of `Ra` to the 2 low-order words of `Ra`, and 2 elements of `Rb` to the 2 high-order words of `Ra`. The elements to be copied are specified by `i`.
`movhlps Ra, Rb`	copy the 2 high-order words from `Rb` to the 2 low-order words of `Ra`.
`movlhps Ra, Rb`	copy the 2 low-order words from `Rb` to the 2 high-order words of `Ra`.

4 One-Pass Vector Sorting

The third SIMD-based sorting algorithm accomplishes the sorting in a single pass. Intuitively this is possible by loading all of the n elements to be sorted into the vector registers, applying the comparators for an n-element (scalar) sorting network, and writing the elements back to memory in-place.

The difficulty with this approach lies in repositioning elements within the vector registers such that the vector comparator operations do not corrupt the values of elements not involved in the comparison. Moreover, simply aligning comparator inputs may be challenging, depending on the fragmentation of free locations within the vector registers.

Since the cost of applying a vector comparator remains the same regardless of the number of "care" values in each input vector, a natural optimization is to execute more than one (scalar) sorting-network comparator at a time. However, the cost of additional data-movement instructions to properly position multiple comparator elements in each vector register may outweigh the benefit of parallelization. Thus, the cost analysis takes into consideration the costs of data movement and comparator instructions for the target architecture.

4.1 Aligning Vector Elements

The sorting network shown in Fig. 1 will be used as a running example in the description of the single-pass in-register sorting algorithm. This network has four vertical rails and requires the execution of five comparison instructions.

An in-register sorting instance of the network of Fig. 1 for the x86-64 SSE(2) SIMD machinery is shown in Fig. 3. The instructions used in this instance are described in Table 1.[3]

The data dependencies in the sorting network define a partial ordering for the execution of the comparisons. The comparators can thus be partitioned into sets in such a way that all the comparators in each set can be executed in parallel. This partition corresponds to the computation of the maximal anti-chains in a data-dependency graph [15]. The sorting

[3]Other SSE2 instructions frequently used for data movement but not included in this example are: `pshufd`, `unpckhps`, and `unpcklps`.

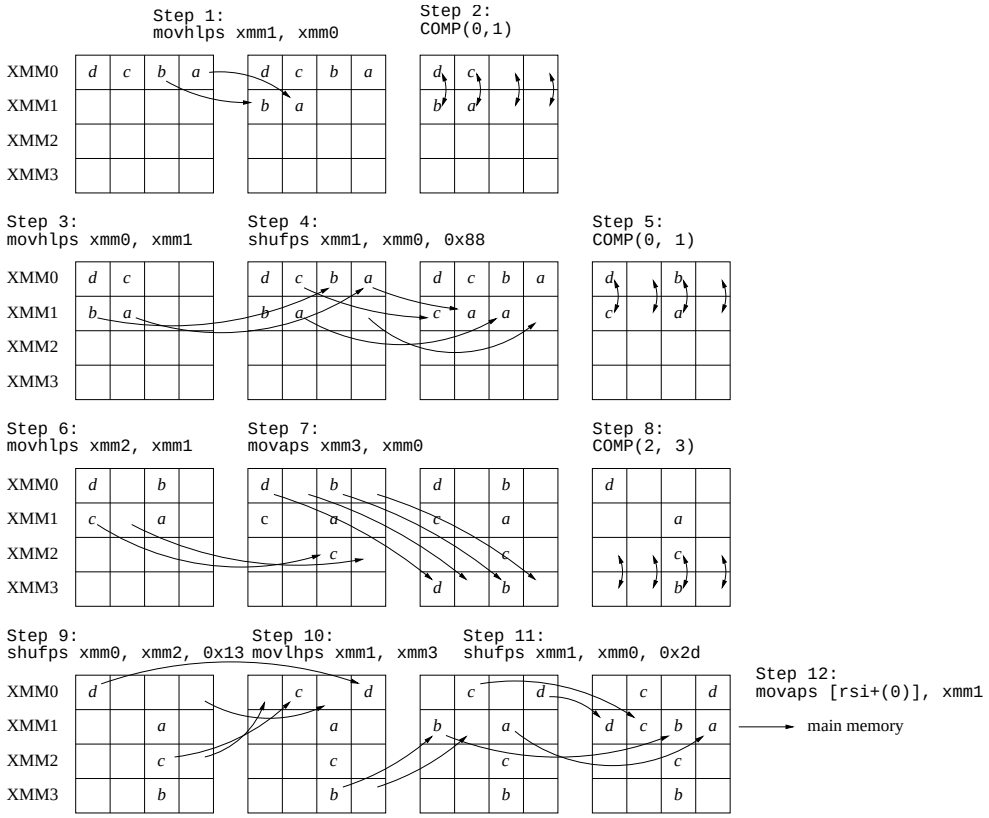

Figure 3: Instruction sequence to apply an in-register 4-element sorting network in an x86-64 architecture. The associated sorting network is shown in Fig. 1.

network of Fig. 1 produces the following partitions: $P_1 = \{COMP(a,c), COMP(b,d)\}$; $P_2 = \{COMP(a,b), COMP(c,d)\}$; and $P_3 = \{COMP(b,c)\}$.

The process of generating the vector instructions for the in-register instantiation of a given sorting network starts with the assignment of each rail to an element of a vector. In Fig. 3 the four rails are assigned the elements of `XMM0`. Then the lowest-cost sequence of vector instructions must be generated to align the rails in order to enable the execution of the comparators in P_1. In this example a single SSE2 instruction, `movlhps` in step 1, aligns b with d and a with c. The comparators $COMP(b,d)$ and $COMP(a,c)$ can then be executed in parallel (step2).[4] After this comparison the value stored in rail b is smaller than the value stored in rail d, and the value stored in rail a is smaller than the value stored in rail c. In Fig. 3 a blank square represents a vector element that contains an unknown value that is not relevant to the sorting process. For instance, after the comparison in step 2 the values that were in rails b and a in the low-order words of `XMM0` may have moved. As they are not part of the sorting process they are now represented by blank squares. If the inputs to the rails are $a = 7$, $b = 2$, $c = 5$, and $d = 9$, this comparison would leave the highest-order words of `XMM0` and `XMM1` intact and would swap the contents of the second highest-order

[4]For SSE2, a comparator between the contents of two registers `Ra` and `Rb` requires a temporary register `T` and the execution of three instructions: `movaps T, Ra`; `minps Ra, Rb`; and `maxps Rb, T`.

words. It may also swap the values in the two low-order words of these registers, but the contents of those words are irrelevant.

Now the two comparators in partition P_2 are candidates for the next vector comparison. A heuristic search is used to find a low-cost sequence of vector instructions to obtain this alignment. The initial state for the search is the position of the rails in the vectors at the end of step 2. The search proceeds by computing the change of state caused by each possible vector instruction that could be applied to this state. Instructions that completely eliminate a given rail from the set of vector registers are discarded. The iterative-deepening search proceeds until a low-cost sequence of instructions that produces the alignments for the comparators in partition P_2 is generated. Separate searches are also conducted for instruction sequences that produce alignments to enable only a subset of the comparators in P_2. At this point the heuristic component of the search decides which sequence of alignment instructions to use. This decision will never be reconsidered. Intuitively the heuristic attempts to select the sequence with the best ratio of number of alignments produced versus instructions required. There is an additional bias towards producing more alignments, since this will reduce the expected number of parallel comparators. In the example in Fig. 3 a sequence of two instructions, `movhlps` and `shufps`, is selected to align rails d with c and b with a. Thus both comparators of P_2 can be executed in parallel in step 5.

A new heuristic search then starts to find the next vector instruction sequence. The scheduling proceeds until all the comparators of the sorting network have been scheduled. After the execution of the last comparator the input values are sorted within the rails, but the rails do not appear in the order in which they must be written back to memory. A similar iterative deepening search now finds the lowest-cost vector instruction sequence to obtain the correct alignment. In Fig. 3 the initial state for this search is the configuration after the comparison in step 8. The final state has the rails aligned in any of the vector registers used for sorting. In this case three instructions are used to align the rails in `XMM1`. And finally `movaps` writes the sorted sequence back to memory.

The vectorization of a sorting network only needs to be done once for each sorting network and for each set of vector instructions. Thus all the heuristic searches described above should be performed once and offline. The resulting schedule can then be instantiated by the code generator whenever a sequence of the corresponding size needs to be sorted.

5 Sorting Key-Pointer Pairs

So far this paper addresses the problem of sorting an array of floating-point values. A more general problem is that of sorting an array of data structures. Consider the case where each structure has a well-defined floating-point key value. Efficient algorithms sort an array of key-pointer pairs to avoid moving large data structures. This section describes an extension of the vectorized sorting networks to handle key-pointer pairs with floating-point keys and a byte sequence representing the pointer.

The solution to the key-pointer sorting problem consists of storing the keys and the pointers into separate SIMD vectors. If keys and pointers appear interleaved in memory, then they must be "swizzled" when loaded into the SIMD vectors and this swizzling must be reversed when storing the sorted result to memory. With the keys and pointers in separate

vectors, the standard sorting network solution is implemented for the keys, while the pointers move in synchrony with the key movements. This is accomplished by using a bitmask to apply the "swap" operations only to selected elements in the pointer vector. Specifically, those elements which correspond to changes in the key vector after applying the key comparator. The construction of this bitmask is supported in architectures that support SIMD operations. For instance, Figs. 11 and 12 show the code used to generate the masks for the sorting-network comparators using the G5 AltiVec intrinsics and the Pentium 4 SSE2 assembly instructions.

6 Vectorizing d-Heaps

d-heaps are a straightforward generalization of binary heaps where each internal node has d children instead of 2. Increasing the value of d results in a shallower tree at the expense of requiring *delete-min* operations to perform more work when searching for the child node with minimum key value. For concreteness we assume that we are dealing with *min*-heaps.

We also assume an implicit heap layout, with all elements stored in a contiguous array. The root node is located at index 0, and the nth child of a node at index i is located at index $i * d + n$, with $1 \leq n \leq d$. The parent of any node may be similarly computed by dividing its index-1 by d. In [8, 9] LaMarca and Ladner investigate the performance of traditional implicit heaps and how they are affected by data caches. They suggest increasing the branching factor d as well as the data alignment techniques described here and used in our implementation.

We present here a method for increasing d-heap performance by using SIMD vector instructions to quickly compute the index of the child with minimum key value. This computation is used within *heapify-down* operations, while *heapify-up* remains unchanged.

This method is similar to the one used for sorting key-pointer pairs in that it relies on the synchronous movement of values within a second set of registers. In this situation the values moving in synchrony are the *indexes* of each child node (specifically the offset from the first child, such that the values range from 0 to $d - 1$).

For simplicity, assume that d is a multiple of k, the number of elements in a SIMD vector. Practically this also lends itself to aligning a node's children on cache-line boundaries. This requires locating the root node at the end of a cache-line such that its first child is at the beginning of a cache-line. This also avoids alignment issues when loading data into SIMD vectors on those architectures which either do not support unaligned accesses, or do so at the cost of increased instruction time.

It is likely that the nodes in the heap are key-pointer pairs, rather than just keys. In this case, loading key values into a SIMD vector may require additional swizzle instructions to interleave the keys from 2 separate vector loads. Only the key values are required; the associated pointers may be discarded.

When a block of keys is loaded into a SIMD vector, the index offsets for those keys are loaded into another SIMD vector. These offsets are simply loaded from a constant and preallocated (*e.g.* static) array with values $0, 1, \ldots, d - 1$. The synchronous movement of the index offsets is implemented in the same manner as the movement of the pointer values in section 5. The loading and movement of these offsets is omitted for clarity.

The algorithm proceeds as follows: (1) load the first k keys into one SIMD vector, call this register A; (2) while unread keys remain, read the next k keys into a SIMD vector B and set $A := min(A, B)$; (3) compare the k values in A against themselves to find the index of the minimum element. If the node being examined does not have d children (this may only occur at last internal node) then the vectorized search is replaced by a straightforward linear scan.

Assembly code for the comparisons in steps (2) and (3) is shown in Figs. 13 and 14. Empirically it was found to be faster for step (3) to compute the index via in-register comparisons rather than writing the final key and index vectors back to memory and comparing those keys using traditional `if` statements.

7 Experimental Evaluation (Sorting)

The three versions of vectorized sorting described in this paper were evaluated by integrating them as the low-level algorithms for the DTSL quicksort. The main findings of this experimental evaluation are:

- all the SIMD-based sorting algorithms for short sequences are more efficient than the SN algorithm shipped with DTSL. The one-pass register sort algorithm requires 5% of the cycles required by SN to sort a 32-input vector;
- the integration of SIMD-based sorting algorithms to sort sequences smaller than a fixed threshold improves the performance of DTSL when sorting floating-point keys by up to 18%;
- this performance improvement is due not only to a reduction in the number of loads, stores, and branch instructions, but also to a significant decrease in the number of branch mispredictions.

7.1 Integrating Algorithms into DTSL

Table 2: Algorithms studied

Algorithm	**Description**
`MSort`X `-` Y	MSort algorithm with X streams applied at Y threshold.
`ISort`X `-` Y	ISort algorithm with X streams applied at Y threshold.
`RSort -` Y	One-pass register sort applied at Y threshold.
`DTSL`	Original DTSL quicksort with SN as the low level algorithm.
`Ins -` Y	Standard insertion sort applied at Y threshold.

The SIMD-based algorithms presented in this paper were integrated in the quicksort implementation of DTSL. The DTSL's quicksort is not recursive. Instead it maintains an in-function stack of current partitions. When the number of elements to be sorted drops below a threshold, DTSL switches to a low-level sorting algorithm. The version of quicksort that produces the best, or close the best, performance when sorting floating-point keys in DTSL uses a scalar sorting network SN as the low-level algorithm [10]. The single-element

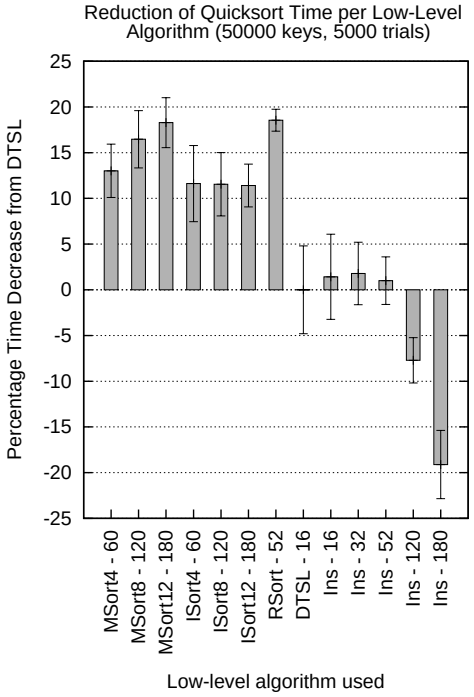

Figure 4: Quicksort wall-clock times relative to DTSL on a 64-bit 3.40 GHz Pentium 4. FP keys.

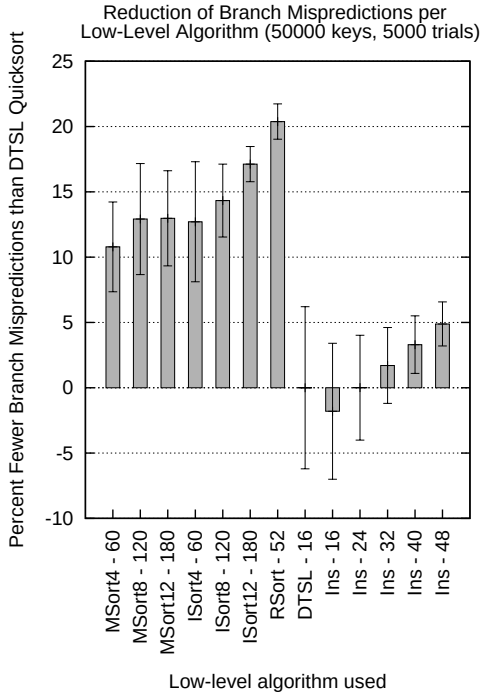

Figure 5: Reduction of branch mispredictions on a 64-bit 3.40 GHz Pentium 4.

comparators in this sorting network are written in the C language and use branch instructions to conditionally perform element interchanges. The default threshold to switch to this low-level algorithm is sixteen elements. This version of DTSL's quicksort is the baseline for the comparative performance study in this paper. Table 2 lists the algorithms used in this performance evaluation. The standard insertion-sort algorithm, `Ins -` Y, is included to provide a familiar comparison point.

7.2 Wall-Clock Execution Time

Experiments were performed on a 64-bit 3.40GHz Pentium 4, an Athlon64 3500+, and an IBM 2.7 GHz PowerPC G5. Fig. 4 shows the relative wall-clock execution times for the sorting of a vector of floating-point keys in relation to the DTSL baseline. Each bar represents the average runtime over 5000 trials on uniformly distributed data. The error bar represents one standard deviation. A similar pattern emerges from the same set of experiments run on the G5. The best performance in the G5 is for `MSort`28`-`840, which results on an average execution-time reduction of 15%. `RSort-`56 is on average 13% faster than DTSL in the G5. The G5 has 32 vector register — instead of the 16 vector registers available in the Pentium 4 — and its AltiVec instruction set offers greater flexibility in the selection of destination registers, which result in RSort requiring fewer permutation instructions with AltiVec than it does with SSE2.

There is no significant change in the pattern of the results when the problem of sorting key-pointer pairs (see Section 5) is solved in both machines.

7.3 Clock Cycles and Branches

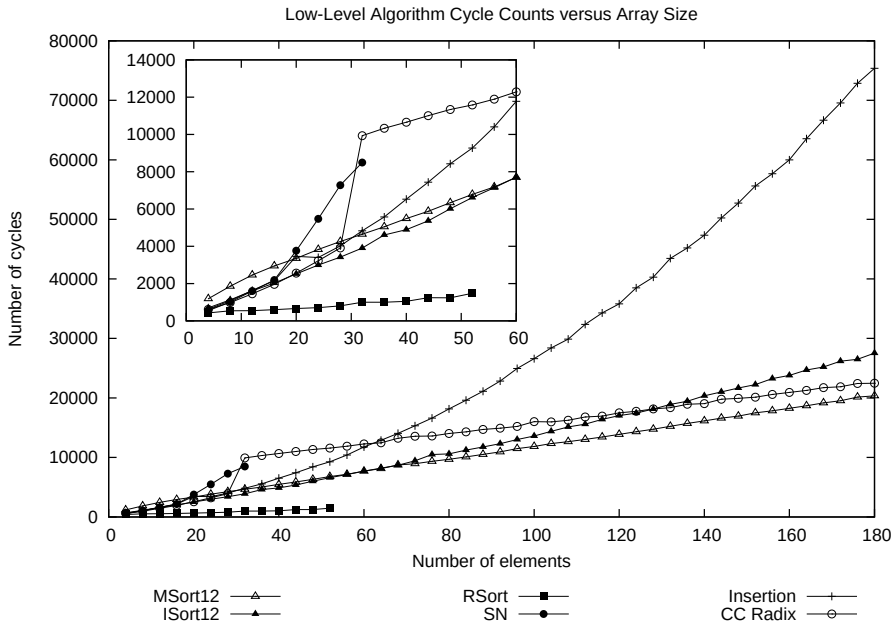

Figure 6: Clock cycles on a 64-bit 3.40 GHz Pentium 4.

Fig. 6 shows the number of clock cycles, obtained through the PAPI library, required by each algorithm as the number of elements to be sorted varies to the maximum possible for each algorithm. Each point in the graph is the average over 10000 trials with uniformly distributed data. This graph shows that RSort is significantly superior to both the SN branch-intensive algorithm and the CC-radix cache-conscious algorithm distributed with DTSL for small sequences, and confirms that MSort is also an excellent choice for the sorting of short sequences. A measurement of the number of branches executed (not shown) confirmed that the elimination of branches is responsible for the superior performance of RSort and MSort.

A detailed study of other performance counters showed a correlation between reduction in the number of branches, loads, and stores executed and the relative performance of the algorithms. One exception was `ISort`12-180 in the Pentium 4. Its wall-clock execution time was on average 12% lower than DTSL, but it performed 5% more loads, 38% more stores, and executed 20% more branches. The explanation, as shown in Fig. 5 seems to be in the reduction in the number of branch mispredictions.

8 Experimental Evaluation (d-Heaps)

The performance of d-heaps was investigated by comparing highly optimized versions with different branching factors against SIMD variants where vector instructions were used dur-

ing *heapify-down* operations. The main findings of this experimental evaluation are:

- using SIMD instructions achieves a 20% reduction in cycle count for a large range of heap sizes, with reductions up to 35%, compared to the best non-SIMD d-heap for each heap size;
- these reductions seem to be tied to a reduction in the number of branch instructions, rather than the number of loads and/or stores;
- the SIMD variants achieve far fewer branch mispredictions than regular d-heaps for values of $d > 2$.

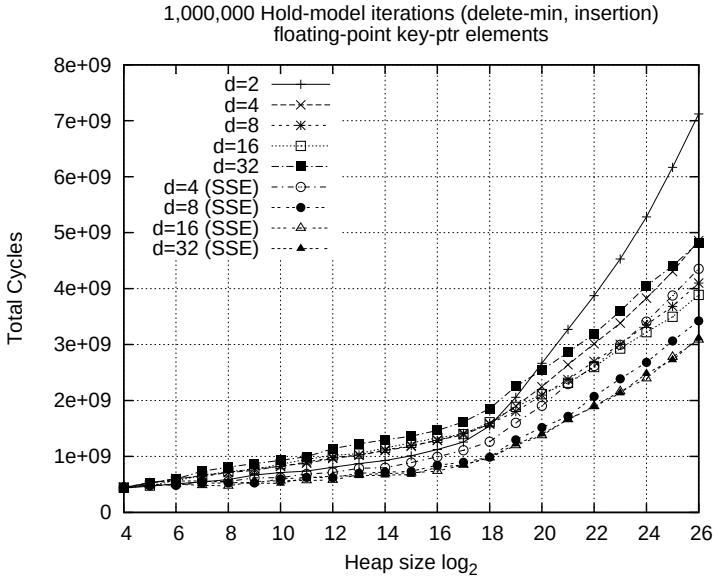

Figure 7: Cycle counts for each heap variant on a 64-bit 3.40 GHz Pentium 4.

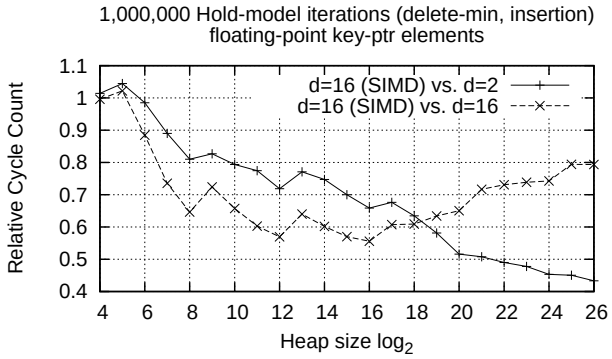

Figure 8: Ratio of the best SIMD d-heap to the 2 best traditional heaps in each range. Run on a 64-bit 3.40 GHz Pentium 4.

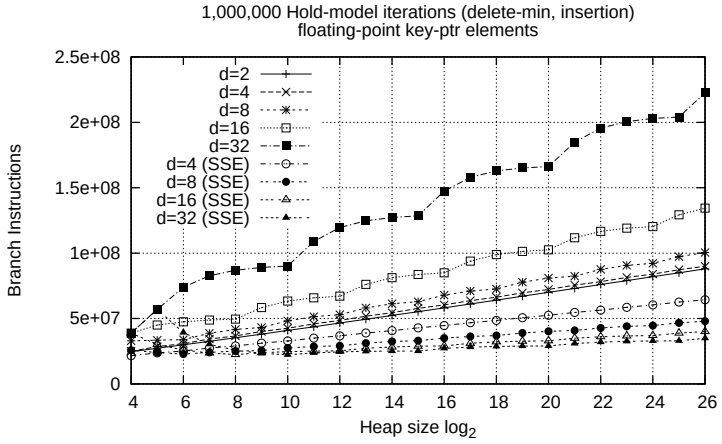

Figure 9: Number of branch instructions on a 64-bit 3.40 GHz Pentium 4.

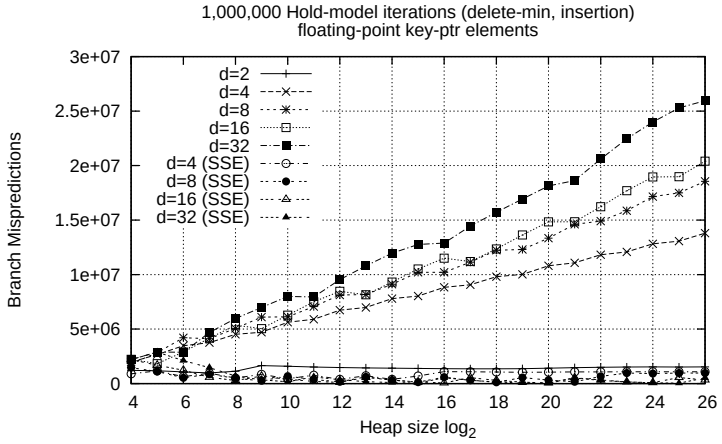

Figure 10: Number of branch mispredictions on a 64-bit 3.40 GHz Pentium 4.

All source code was written in C++ and was compiled using g++ 3.4.6, with full optimizations and loop unrolling. The branching factor d was known at compile time. The heap itself was aligned in memory such that the root node's children began on a cache-line boundary. A consequence is that all children are aligned for SIMD vector accesses. The binary heap had further optimized index computations.

Experiments were performed on a 64-bit 3.40GHz Pentium 4. Heap elements are key-pointer pairs with floating point keys. Heap sizes are powers of 2, from 2^4 to 2^{26}. Heaps are initialized by inserting n elements, where n is the maximum size of the heap. Keys for initial elements are drawn uniformly from $0, \ldots, n-1$.

1,000,000 iterations of the "hold" model were then performed. Each iteration consists of a call to *delete-min* followed by *insert-element*. The key of the new element is equal to the key of element last removed plus a value drawn uniformly from $0, \ldots, n-1$.

As seen in Fig. 7, when the heap size becomes 2^{18} there is a crossover between values of d in the performance of traditional heaps. For small heaps $d = 2$ performs better, while $d =$

16 performs better for larger heaps, resulting from better locality of each node's children as well as decreased heap depth. For the SIMD d-heaps, $d = 16$ seems to consistently perform the best, although almost identical to $d = 32$.

Fig. 8 shows the ratio of execution times between this best SIMD heap versus the two best values of d for traditional heaps. For a large range of values ($n \geq 2^{10}$) the SIMD version executes in at least 20% fewer cycles, up to an approximate 35% reduction in cycles at $n = 2^{18}$.

Figs. 9 and 10 show the number of branch instructions and mispredictions respectively. The branch instructions seems to closely track SIMD d-heap cycle counts, although in general performance is not directly attributable to branches, mispredictions, or stores and loads. For all heap sizes the traditional binary heap exibited a low and essentially constant number of branch mispredictions, with all the SIMD variants exibiting slightly fewer.

9 Related Work

The implementation of sorting in large-scale vector machines has been extensively studied. Siegel produced one of the earliest descriptions of how to implement Batcher's sorting network, also known as *bitonic sorting*, in SIMD machines [14]. Bitton *et al.* provides an extensive description of such implementations [3]. The new contribution of this paper is to demonstrate how the well-known sorting networks can be implemented in the SIMD machinery of contemporary processors and to indicate that code generators can instance such implementations to improve the performance of recursive sorting algorithms and heaps.

The idea of making better use of register resources within the processor to reduce the number of load of stores, in our case to put the SIMD resources to good use in sorting, is also explored by Arge *et al.* [17]. Their idea of forming cache-load-sized runs with quicksort is similar to our idea of switching to SIMD-register-based sorting at an appropriate threshold. The contrast is that we are also benefiting from the SIMD machinery which allows more parallelism in the execution and the elimination of branches while they use the general-purpose registers and the storage available at a cache line.

Recently compilers have been used more often to improve the code generation for SIMD machinery in contemporary processors. Ren *et al.*'s approach of using an optimization algorithm to improve the data permutations is more general than our specific iterative-deepening search [13]. Nuzman *et al.* describes a compiler framework to generate vectorized code for interleaved data [12].

The relationship between the SIMD-register-based sorting algorithms presented in this paper and the development of DTSL is an orthogonal improvement to a library generator [10]. Li *et al.* focused on the dynamic identification of the best sorting algorithm for a given input sequence [11]. They selected an efficient algorithm for the tail of their recursive method. This paper offers a better solution for the sorting of sequences that are small enough to benefit from the use of the SIMD machinery. Similarly, we provide a faster mechanism for selecting a minimum (maximum) child in the implicit d-heaps studied by LaMarca and Ladner [8, 9].

10 Conclusions

This paper proposes the use of the SIMD machinery provided in modern processors to improve the performance of recursion tails. The idea is that whenever the number of elements to be processed fits within the SIMD registers available in the processor, these values should be loaded once into the SIMD registers and then an efficient SIMD execution should be used. While the feasibility of this idea was demonstrated with the integration of a more efficient algorithm for sorting short sequences into DTSL, the idea should be generally applicable to recursive computation.

Once efficient low-level SIMD algorithms are crafted, they can be generated into a solution database to be instantiated by code generators into optimized libraries. Alternatively, if a suitable identification algorithm is created, the compiler should be able to integrate these solutions directly into general programs.

Acknowledgments

The experimental evaluation of these ideas was made possible thanks to David Padua's generous sharing of his group's DTSL code. This research is support by grants from the Natural Science and Engineering Research Council (NSERC) of Canada, and by IBM Corporation.

References

[1] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[2] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, 1998.

[3] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):287–318, September 1984.

[4] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Principples and Practice of Parallel Programming PPoPP*, pages 206–216, Las Vegas, Nevada, 1997.

[5] M. Frigo. A fast Fourier transform compiler. In *Programming Language Design and Implementation PLDI*, pages 169–180, Atlanta, GA, June 1999.

[6] Intel. IA-32 Intel®architecture software developer's manual volume 1: Basic architecture. http://download.intel.com/design/Pentium4/manuals/25366519.pdf, 2006.

[7] Donald Ervin Knuth. *The Art of Computer Programming, Vol. 3 - Sorting and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1973.

[8] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithms*, 1:4, 1996.

[9] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[10] X. Li, M. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Code Generation and Optimization CGO*, pages 111–122, Palo Alto, CA, 2004.

[11] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithms. In *Code Generation and Optimization CGO*, pages 99–110, San Jose, CA, March 2005.

[12] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Programming language design and implementation PLDI*, pages 132–143, 2006.

[13] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for SIMD devices. In *Programming language design and implementation PLDI*, pages 118–131, 2006.

[14] H. J. Siegel. The universality of various types of SIMD machine interconnection networks. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 23–25, Silver Spring, MD, March 1977. ACM SIGARCH/IEEE-CS.

[15] S. A. A. Touati. Register saturation in instruction level parallelism. *International Journal of Parallel Programming*, 33(4):393–449, 2005.

[16] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of sotware and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[17] R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithmics*, 7:9, 2002.

[18] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Programming Language Design and Implementation PLDI*, pages 298–308, Snowbird, Utah, June 2001.

A Appendix

```
inline void COMP_VEC_PTR(vector float &key_a, vector float &key_b,
                         vector int   &ptr_a, vector int   &ptr_b)
{
  vector float temp = key_a;                // move the low key-values
  key_a = vec_min(key_a, key_b);            // into key_a, and the high
  key_b = vec_max(key_b, temp);             // values into key_b

  vector int mask = vec_cmpeq(temp, key_a);  // compare the original values of key_a with
                                             // the new values, to see which have moved
  vector int temp2 = ptr_a;
  ptr_a = vec_sel(ptr_b, ptr_a, mask);       // use the bitmask to select the appropriate
  ptr_b = vec_sel(temp2, ptr_b, mask);       // values for the pointer vectors
}
```

Figure 11: Key-pointer comparator using the G5 AltiVec vector intrinsics.

```
asm("pshufd  xmm15, xmm1, 0xE4");   // xmm15 := copy of key_a (xmm1)
asm("minps   xmm1,  xmm2");         // key_a := minimum(key_a, key_b)
asm("maxps   xmm2,  xmm15");        // key_b := maximum(key_b, orig_key_a)

asm("cmpps   xmm15, xmm1, 4");      // xmm15 := bitmask (key_a != orig_key_a)
asm("pshufd  xmm14, xmm3, 0xE4");   // xmm14 := copy of ptr_a
asm("xorps   xmm14, xmm4");         // q     := ptr_a XOR ptr_b
asm("andps   xmm15, xmm14")         // q     := q AND bitmask
asm("xorps   xmm3,  xmm15");        // ptr_a := ptr_a XOR q
asm("xorps   xmm4,  xmm15");        // ptr_b := ptr_b XOR q
```

Figure 12: Key-pointer comparator using the Pentium 4 SSE2 assembly instructions. Vector registers xmm1 and xmm2 hold keys, registers xmm3 and xmm4 hold the respective pointers. Registers xmm14 and xmm15 are used as temporary storage.

```
asm("minps   xmm1,  xmm2");        // xmm1 := minimum(key_a, key_b)
asm("cmpps   xmm2,  xmm1, 0x04");  // mask := bitmask (key_b != orig_key_b)
                                   //                 (key_a == orig_key_a)
asm("andps   xmm3,  xmm2");        // idx_a    := idx_a & mask
asm("andnps  xmm2,  xmm4");        // idx_b    := idx_b & ~mask
asm("orps    xmm3,  xmm2");        // idx_a    := idx_a | idx_b
```

Figure 13: Selecting the minimum key and its associated index in parallel using the Pentium 4 SSE2 assembly instructions. Vector registers xmm1 and xmm2 hold keys, registers xmm3 and xmm4 hold the respective indexes.

```
// compare the upper vector elements against the lower
{
asm("movhlps  xmm2, xmm1");
asm("movhlps  xmm4, xmm3");

// select_min(key_a=xmm1, key_b=xmm2, idx_a=xmm3, idx_b=xmm4)
// will invalidate b=1 in the process
asm("minps   xmm1, xmm2");
asm("cmpps   xmm2, xmm1, 4"); // neq
asm("andps   xmm3, xmm2");    // idx_a := idx_a & mask
asm("andnps  xmm2, xmm4");    // idx_b := idx_b & ~mask
asm("orps    xmm3, xmm2");    // idx_a := idx_a | idx_b
}

// compare the two lower vector elements
{
asm("pshufd   xmm2, xmm1, 0x01"); // 00 00 00 01
asm("pshufd   xmm4, xmm3, 0x01"); // 00 00 00 01

// final_select_min(key_a=xmm1, key_b=xmm2, idx_a=xmm3, idx_b=xmm4)
// ONLY correctly updates idx_a (not key_a or idx_b)
// will invalidate key_b in the process
asm("cmpps   xmm1, xmm2, 0x01");  // mask := key_a < key_b
asm("andps   xmm3, xmm1");        // idx_a    := idx_a & mask
asm("andnps  xmm1, xmm4");        // idx_b    := idx_b & ~mask
asm("orps    xmm3, xmm1");        // idx_a    := idx_a | idx_b
}

// move the index of the minimum key into local variable "final"
asm("movss   [%0], xmm3" : : "r" (final) );
```

Figure 14: SSE2 data movement and comparison instructions to extract the index of the minimum key in one vector. Vector register xmm1 holds keys, vector register xmm3 holds the respective indexes.