AN AUTOMATICALLY-TUNED SORTING LIBRARY

ERAN BIDA AND SIVAN TOLEDO
SCHOOL OF COMPUTER SCIENCE, TEL-AVIV UNIVERSITY

Abstract

We present ATSL, an automatically-tuned sorting library. ATSL generates an in-core sorting routine optimized to the target machine for a specific data type. ATSL finds a high-performance sorting routine by searching an algorithmic space that we have defined. The search space includes basic sorting algorithms and automatically-generated compositions of sorting algorithms. Performance measurements are used both for ranking candidate algorithms and for characterizing the behavior of candidates in specific settings (ranges of array sizes). These characterizations allow ATSL to generate hybrid algorithms that intelligently exploit the strengths of particular algorithms, such as high speed at specific input-size ranges. Many sorting algorithms can be tuned using numeric parameters. ATSL searches these parameter spaces to find values that yield high performance on the target machine. The building blocks from which ATSL synthesizes sorting algorithms include adaptations of many of the most effective hand-tuned sorting routines, including several that are tuned for cache efficiency.

An extensive experimental evaluation shows that ATSL generates high-performance codes that are well tuned for the target machine and data type. The experiments were conducted on six different machines, of several architectures, and with three different compilers. The algorithms that are generated are fast; in particular, they beat the hand-tuned building blocks and the compiler's C++ built-in sorting routine. The algorithms that ATSL generates on different machines and using different compilers are different from each other.

1. Introduction

We describe ATSL, a system for creating automatically-tuned sorting libraries. Given a data type, ATSL synthesizes a large number of sorting algorithms for that type, measures their performance on the target machine, and constructs a high-performance sorting routine.

Our main goals in designing ATSL were to create sorting algorithms that would match and beat the best hand-coded sorting libraries, and to create a highly-usable system. We were less interested in other aspects of the problem, such as adapting to specific inputs (beyond their size) and efficient searching of the

parameter space. These aspects are important, but they were already explored by others [15, 16].

In the last few years, automatically-tuned libraries have been successfully developed for several fundamental computations. ATLAS [24] and PHiPAC [2] are automatically-tuned libraries for dense-matrix kernels (e.g., matrix multiplication); OSKI [23] generates sparse-matrix kernels; FFTW [8] and UHFFT [18] are automatically-tuned FFT libraries. Some of these systems, like ATLAS and FFTW are already widely used in practice. The problems that have been addressed by these systems share two common features: the problems are important, so they deserve the software engineering effort that goes into such a system, and there is a substantial literature on algorithms to solve them. In virtually all cases, the literature not only describes multiple algorithms, but also many architecture-dependent and -independent optimizations, tuning techniques, and comparisons. In other words, automatic tuning systems exploit years or decades of research on tuning by hand.

Sorting belongs to the same category of problems, but until recently, there were no automatically-tuning systems for sorting algorithms. Sorting is important, and sorting algorithms have been investigated for at least half a century. In particular, several architecture-dependent optimizations, such as cache-efficient algorithms, have been investigated in the last few years. Therefore, we set out in June 2003 to develop a system for synthesizing automatically-tuned sorting libraries. We were especially interested in exploiting, matching, and beating hand-crafted cache-optimized sorting techniques.

Recently, Li, Garzarán and Padua presented a similar system [16], which appears to produce algorithms of similar quality. Our system is quite similar to theirs, and our main conclusions match theirs: that an automatic tuning of sorting algorithms is viable and can produce better results than hand-coding. However, our approach is somewhat different than theirs. The three most important differences between our results and theirs are that we propse a formal and abstract definition of the search space, that our system is designed to be open and to easily accomodate new algorithms and implementations, and that our results show that automatic tuning is superior to virtually all the hand-tuned codes produced in the last decade or so. We discuss these differences in more detail in Chapter 8.

The rest of this paper is organized as follows. We survey relevant related work in the next chapter. The description of ATSL spans Chapters 3 through 6. ATSL constructs an efficient algorithm by searching a large space of potential candidates. These candidates are mostly synthetic algorithms that are automatically generated from hand-coded building blocks and from hand-coded templates. Chapter 3 describes this search space. More specifically, it describes a notation, a naming scheme for sorting algorithms. Essentially, ATSL searches this name space. To evaluate an algorithm in this space, ATSL must synthesize it. Chapter 4 explains how ATSL synthesizes a given algorithm. Chapter 5 describes how

ATSL searches the space and how it compares candidate algorithms. The design and implementation of ATSL are described in Chapter 6. Experimental results, which show the effectiveness of ATSL (but also one weakness) are presented in Chapter 7. We present our conclusions from this research in Chapter 8.

2. Related Work

Three categories of published research are relevant to our research. Obviously, one category consists of papers on specific sorting algorithms. Another category consists of papers on adapting sorting algorithms to specific architectural features, such as cache memories. The last category consists of papers on systems for automatically producing highly-tuned algorithms for various problems, not necessarily sorting.

The literature on sorting algorithms is vast. We do not attempt to cite all the relevant papers here. What we do mention are the most important papers on the main algorithms that ATSL uses, and surveys and textbooks that cite additional works. Mergesort, radixsort, insertionsort and bubblesort appear to be folk algorithms from the pre-computer era. Heapsort was invented by Williams [26]. Quicksort was invented by Hoare [10]. Several of the variants of quicksort that we mention are due to Sedgewick [20].

A few monographs are devoted mostly or exclusively to sorting, including Knuth's classic monograph [13], Akl's monograph on parallel sorting algorithms [1], and Mahmoud's monograph on probabilistic analysis of sorting algorithms [17]. Most general textbooks on algorithms, such as [3, 12, 21], describe all the basic sorting algorithms.

Architecture-specific design and optimization techniques for sorting algorithms have been proposed by many authors. Many of these have focused on two fundamental architectural issues, parallelism and cache efficiency. ATSL produces only sequential algorithms, so the parallelism issues are less relevant to us. LaMarca and Ladner investigated cache-efficient variants of several sorting algorithms [14]. Wickremesinghe, Arge, Chase and Vitter proposed a variant of mergesort optimized for caches and for processors with large register files [25]. Sanders and Winkel proposed a cache-efficient samplesort.[19]. ATSL uses algorithms from these three sources. Cache-oblivious algorithms exploit cache memories, but in theory at least they do not need architecture-specific tuning. Cache-oblivious sorting algorithms were first developed by Frigo, Leiserson, Prokop, and Ramachandran [9]. Brodal et al. [5, 6] and Youn [27] describe implementations of cache-oblivious sorting algorithms. We have tried Youn's implementation and found it to be much slower than other sorting algorithms that are included in ATSL. These findings are consistent with Youn's own findings.

Automatically-tuned libraries have already been mentioned in the Introduction. ATLAS [24] and PHiPAC [2] are automatically-tuned libraries for dense-matrix kernels (e.g., matrix multiplication); OSKI [23] generates sparse-matrix kernels;

FFTW [8] and UHFFT [18] are automatically-tuned FFT libraries. All of these systems tune codes by automatically searching a certain space of potential algorithms for efficient ones. Usually, the space is the same space that developers have been searching manually when hand-tuning their codes. In general, all of these systems bring decades of experience in hand-tuning into a problem-specific automatic tuning and optimization system.

Li, Garzarán and Padua [15, 16] describe, XSORT, an automatic-tuning system for sorting. The main difference between ATSL and their system is that XSORT measures the entropy of the input array at run-time and attempts to select an algorithm that works well in that entropy range. The algorithm that ATSL uses depends on the size of the input array, but not on its contents. The two systems also differ in the search algorithm that they use. The two systems were developed concurrently and independently.

3. A Notation for Synthetic Sorting Algorithms

ATSL synthesizes efficient sorting algorithms using parametrization and using two composition rules. We now explain the synthesis principles, which drive most of the design of ATSL. We also describe a notation for synthetic algorithm. The rest of the paper uses the notation to describe ATSL. The most important aspect of the notation is that it is static: a name denotes an algorithm, not an execution of an algorithm. This may seem obvious, but due to the recursive nature of many sorting algorithms, and to the fact that one algorithm may call a different one, developing an appropriate static notation is not trivial. Our notation is similar to the ones presented by Li et al. [16], but in their papers the distinction between the static notation and the dynamic execution is not clear. In particular, they do not clearly define how a static name and an sorting code relate to each other.

The two composition rules are *divide-and-conquer* composition and *size-selection* composition. In divide-and-conquer composition, one sorting algorithm calls another to sort sub-sequences. For example, mergesort divides the input into sub-sequences, recursively sorts each one, and merges the sorted runs. Mergesort can invoke itself to sort the sub-sequences, but it can also call another, arbitrary, sorting algorithm to sort them. We can compose mergesort with, say, insertion-sort, by synthesizing a mergesort procedure that calls insertionsort to sort the sub-sequences. We denote this composition by MERGESORT(INSERTIONSORT). More generally, the notation $r(f)$ denotes a divide-and-conquer algorithm r that calls a sorting algorithm f to sort sub-sequences. Since ATSL synthesizes code, the names in the notation, such as MERGESORT, denote an actual implementation of a sorting algorithm, not the abstract sorting method.

Divide-and-conquer composition is a recursive process. A divide-and-conquer sorting code can call another divide-and-conquer code, which in turn calls a third algorithm. For example, if a mergesort code calls a quicksort code which calls an insertion sort, the notation is MERGESORT(QUICKSORT(INSERTIONSORT)).

Clearly, a code can call itself, as in MERGESORT(MERGESORT(INSERTIONSORT)). However, if a code *always* calls itself, we do not denote this recursive call as a composition, because the recursive call does not allow an algorithm-synthesis system to compose a variety of algorithms.

Our notation denotes static sorting codes, not dynamic executions. In particular, MERGESORT(MERGESORT(INSERTIONSORT)) is a specific synthetic code. On the other hand, a merge sort algorithm that always calls itself recursively is denoted MERGESORT, not

$$\underbrace{\text{MERGESORT}(\text{MERGESORT}(\cdots(\text{MERGESORT})\cdots))}_{\lfloor \log_2 n \rfloor + 1 \text{ times}} \qquad \text{(this is wrong)}.$$

This is not a correct use of the notation because the $\lfloor \log_2 n \rfloor + 1$ invocations of MERGESORT represent the dynamics of a specific execution, not a static composition of codes. With a larger n, the recursive invocation in this particular code would be deeper, but we want the notation to denote the code, not the input-dependent execution.

We use the term *leaf algorithms* to refer to sorting codes that do not call other sorting codes.

Some divide-and-conquer algorithms sort two different kinds of sub-sequences. Consider SAMPLESORT, for example. This algorithm selects a sample $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of k input elements and sorts the sample into $a_{\pi_1} \le a_{\pi_2} \le \cdots \le a_{\pi_k}$. Now SAMPLESORT partitions all the input numbers into $k+1$ disjoint subsets $S_1, \ldots, S_{k+1}$, such that elements in S_j are at least $a_{\pi_{j-1}}$ and at most a_{π_j} (using $a_{\pi_0} = -\infty$ and $a_{\pi_{k+1}} = \infty$). The subsets are now sorted and concatenated to form the output. SAMPLESORT can use the same sorting algorithm to sort both the sample and the subsets, but it can also invoke two different algorithms for the two kinds of sub-problems. We denote such compositions by $r(f_1, f_2)$, where f_1 is the algorithm used for the first kind of sub-problem and f_2 for the second kind. For example, SAMPLESORT(INSERTIONSORT, MERGESORT) denotes a SAMPLESORT procedure that calls INSERTIONSORT to sort the sample but calls MERGESORT to sort the subsets of the input. (Examples like SAMPLESORT suggest that the name *divide-and-conquer* is perhaps not appropriate; however, an alternative term that we have considered, recursive composition, is also misleading because one sorting algorithm can call another to sort subsequences; this is not a recursion.)

The second composition rule that we use is size selection. A *size selector* is a sorting algorithm that sorts the input by invoking one of several other sorting algorithms, one for each range of input sizes. We denote a size selector by a vector of pairs $[(n_1, s_1), (n_2, s_2), \ldots, (n_k, s_k)]$. The first entry in each pair is a positive integer (except n_k which is always ∞) and the second is a sorting algorithm. The selector invokes s_1 on inputs of size $0 \le n < n_1$, invokes s_2 on inputs of size $n_1 \le n < n_2$, and so on. When the thresholds $n_1, n_2, \ldots, n_{k-1}$ are clear from the context, we drop them and denote the selector by $[s_1, s_2, \ldots, s_k]$.

We use the name CALLER to denote the caller of a sorting algorithm. This notation is used mostly in size selectors. For example,

$$\text{MERGESORT}([(31, \text{INSERTIONSORT}), (\infty, \text{CALLER})]) \tag{3.1}$$

denotes a mergesort code that calls itself on sub-sequences of 31 elements or more, but calls an insertionsort code on shorter sub-sequences. The name CALLER is used here in the context of a size selector. Therefore, it denotes the caller of the selector, which is MERGESORT. This is a different algorithm from

$$\text{MERGESORT}([(31, \text{INSERTIONSORT}), (\infty, \text{MERGESORT}')]) \ , \tag{3.2}$$

in which one mergesort (named MERGESORT) calls an insertionsort on short sub-sequences and another mergesort implementation (named MERGESORT′) on long sub-sequences. In particular, if we run the code denoted by (3.1) on an array of size 1000, it will eventually invoke INSERTIONSORT, but if we run the code denoted by (3.2) on an array of size 1000, it will partition the array and it will call MERGESORT′ on the two halves; but MERGESORT′ will sort the halves itself. Let us consider another example, to further illustrate the notation:

$$[(31, \text{INSERTIONSORT}), (\infty, \text{MERGESORT}(\text{CALLER}))] \ . \tag{3.3}$$

This is again different from (3.1). When invoked on an array of size 30, the algorithm (3.1) partitions the input into sub-sequences of 15 elements each (it is a mergesort variant), and calls a the size selector to sort the halves. The size selector calls INSERTIONSORT to sort each half. On the other hand, on the same input (3.3) calls INSERTIONSORT on the entire input, because the top-level algorithm (the outermost algorithm in the expression) is the selector. On larger inputs, it will invoke MERGESORT, which will invoke the size selector to sort the sub-sequences. *The* CALLER *notation allows us to assign a fixed-size name to a recursive algorithm, in which the depth of the recursion may be unbounded. Therefore, the* CALLER *notation is a critical element in the definition of the search space of sorting algorithms.*

There are other possible selectors, but ATSL uses only size selectors. Li et al. describe a system that uses entropy selectors, for example [16].

Many sorting algorithms have tuning parameters. To fully specify an algorithm, these parameters must be specified. We give several examples. A multiway-mergesort partitions the input into k sub-sequences, sorts them, and merges the k sorted runs; k is a parameter. Samplesort selects a sample of k input elements, partitions the input into mutually-monotonic subsets, sorts the subsets, and concatenates them; k is a parameter. Radixsort sorts the input by iterating over groups of k bits of the keys; the radix 2^k is a parameter. Quicksort selects a pivot to partition the input. If the pivot is the median of k random

elements, then k is a parameter. Here are a few examples:

$$\text{MULTIWAY-MERGESORT}_{\text{multi=14}}$$
$$\text{SAMPLESORT}_{\text{sample-size=7}}$$
$$\text{RADIXSORT}_{\text{radix=16}}$$
$$\text{QUICKSORT}_{\text{median-of=3}}\,.$$

Some sorting algorithms solve sub-problems that are not sorting problems. Consider quicksort, for example. It solves two kinds of non-sorting sub-problems. One is the selection of a pivot, and the other is the separation of elements smaller than the pivot from the rest. There are several ways to solve each sub-problem. The pivot can be selected using a deterministic linear-time median-finding algorithm, or it can be the median of a small random sample of the elements, or it can simply be the first element. The partitioning can be done in at least two different ways.

There are several ways to name concrete sorting codes of this type. We can view each combination of sub-problem solvers as a different sorting algorithm. A quicksort variant that invokes a deterministic median-finder and a Lomuto partition might be called QUICKSORT-DETMEDIAN-LOMUTO. An alternative is to view the concrete code as a result of a specialized composition process that composed the top-level algorithm with specific algorithms to solve sub-problems. The notation for this view might be $\text{QUICKSORT}(\ldots, \text{DETMEDIAN}, \text{LOMUTO})$, where the ellipsis denote some sorting algorithm that is called recursively. Finally, we can view the final code as a parametrization of a generic algorithm, for which the notation is $\text{QUICKSORT}_{\text{pivot=DETMEDIAN,partition=LOMUTO}}$.

There is no compelling reason to favor one of this viewpoint over the others. We use the following guidelines. When the construction of the final code is done by hand, we view each variant as a separate code and give them separate names. This emphasizes that we do not inform ATSL in any way that the codes are somehow related. When the construction of the final code is done by a code generator, we view the process as composition or parametrization (either notation is reasonable), to emphasize that an automatic code generator is used. Since these compositions are always specific to a particular sorting algorithm (sometimes to a particular implementation), such a code generator is always algorithm-specific. In contrast, the composition of sorting algorithms can be and is performed by a generic code generator.

4. Generating a Sorting Algorithm from an Abstract Specification

The notation that we developed in the previous chapter allows us to name a wide variety of sorting algorithms. ATSL searches this name space for an efficient algorithm. The search, and also the production of the output algorithm, require the generation of algorithms that correspond to particular names. In this

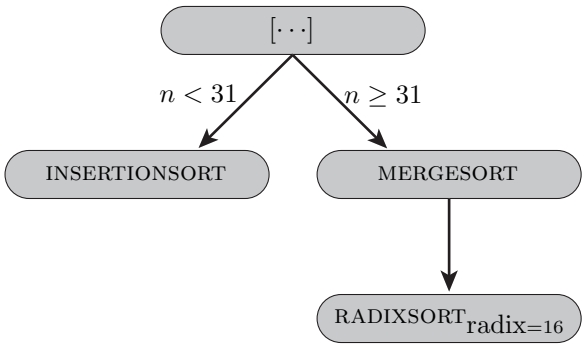

FIGURE 4.1. The expression tree for $[(31, \text{INSERTIONSORT}), (\infty, \text{MERGESORT}(\text{RADIXSORT}_{\text{radix=16}}))]$.

chapter we explain how ATSL generates the code that corresponds to a name like $[(31, \text{INSERTIONSORT}), (\infty, \text{MERGESORT}(\text{RADIXSORT}_{\text{radix=16}}))]$.

The code generation is performed bottom up on the expression tree of a name. Consider the expression tree shown in Figure 4.1. The generation starts at the leaves, INSERTIONSORT and $\text{RADIXSORT}_{\text{radix=16}}$. One leaf, INSERTIONSORT, is not parametrized, so presumably it is a hard-coded implementation that requires no generation. The other leaf is parametrized, so ATSL generates the RADIXSORT variant with the given parameter. Now that the code for the leaves is available, ATSL can generate the composition of MERGESORT with the RADIXSORT leaf. This leaf was generated with a unique function name, say `radixsort_radix_16` (the actual names in ATSL are different). The mergesort variant that ATSL generates calls this particular function. Now that the composed MERGESORT is ready, ATSL generates the size selector, which ends the generation process for this specification.

This high-level description hides some important details that we describe next.

4.1. **Parametrization and Composition using Macros.** ATSL uses the C macros and C++ templates to perform most of the parametrization and composition. We explain the use of C macros in detail. The use of C++ templates is completely analogous. In C, algorithms are implemented as parametrized macros that expand to C functions. Figure 4.2 shows an example of such a macro. Each instantiation of this macro, say `QUICKSORT_MEDIANOF_M_F(3,insertionsort)`, generates a particular parametrized composition of the code.

The instantiation of parameters and callee names at compile time serves two purposes. First, it is more efficient at run-time than passing the parameters (and function pointers) to a generic sorting function. Second, it allows the compiler a much wider scope for optimization. In the example, the compiler can use constant-folding to completely unroll the loop that select the pivots and the loops that select the median of the sample (the median selection code is not shown in Figure 4.2). The compiler can also perform inter-procedural optimizations on

```
#define QUICKSORT_MEDIANOF_M_F(M,F)                                  \
  void quicksort_medianof_##M##_##F##(datatype* A, int n) {          \
    int      pivots[ M ], i, j;                                      \
    datatype pivot;                                                  \
    if (n <= 1) return;                                              \
    for (i=0; i<M; i++) pivots[i] = A[ random_int(0,n-1) ];          \
    ... /* inline median finding                        */          \
    ... /* partitioning into A[0..j-1] and A[j..n-1] */             \
    F(A  , j  );                                                     \
    F(A+j, n-j);                                                     \
  }
```

FIGURE 4.2. A macro that generates $\text{QUICKSORT}_{\text{median-of=}m}(f)$.

`quicksort_medianof_3_insertionsort` and on `insersionsort` together, or it might inline `insertionsort`.

4.2. **Parametrization by Code Generation.** Binding parameters at compile time gives the compiler a wide scope for optimizations, but whether the compiler actually performs them or not depends on the particular compiler. In some cases, researchers have shown that explicit production of optimized sorting code is necessary. For example, Wickremesinghe et al. [25] found it effective to generate straight-line code for priority-queue operations with a fixed-size heap. The straight-line code allowed them to use scalar variables for the elements of the heap, rather than an array. The use of scalar variables allowed the compiler to assign them to registers.

To support such cases, ATSL supports specialized code generators for parametrized algorithms. We use this mechanism to generate Wickremesinghe-style heaps for multiway mergesort. We also use this mechanism to generate radixsort algorithms for specific radixes.

Similar mechanisms are used in automatic code generators/tuners for other problems. For example, hand optimization was common in matrix algorithms. A code generator, ATLAS, generates similarly optimized source codes automatically. Our experience with ATLAS has been that explicitly generating optimized source code is helpful when the compiler's optimizations are not aggressive enough, e.g., under GCC versions 2 and 3, but is not helpful with more aggressive optimizing compilers, such as SGI's [11]. FFTW, a code generator for fast Fourier transforms, generates code that incorporates FFT-specific optimizations, which exploit symmetries in the FFT algorithm.

4.3. **Limitations I: Caller Equivalences.** ATSL supports the CALLER notation only in specific contexts. General support for CALLER would require either passing a function pointer to the caller, or generating caller-specific functions.

We first describe where ATSL support the CALLER notation and how, and then describe how it might be supported in a general way.

ATSL supports the CALLER notation only in subexpressions of the form

$$[(t, f), (\infty, r(\text{CALLER})] \ .$$

Such an expression corresponds to a sorting routine that calls f on arrays smaller than t and r on larger arrays, where r calls its caller (the binary size selector) to sort sub-arrays. Both f and r might have additional parameters. To support such expressions, for each possible r we create a parametrized algorithm r' in which the CALLER is eliminated. More specifically, $r'_{\text{threshold}=t}(f)$ is an algorithm that sorts by calling f on inputs smaller than t; on larger inputs, it calls $r' = r(r'_{\text{threshold}=t}(f))$, a variant of the recursive algorithm r in which the recursive call is statically bound to r', not to CALLER $= [(t, f), (\infty, r(\text{CALLER})]$. It is easy to see that the two are equivalent,

$$[(t, f), (\infty, r(\text{CALLER})] \equiv r'_{\text{threshold}=t}(f) \ .$$

The equivalence follows from the following inductive argument. On inputs of size $\tilde{t} < t$, both algorithm invoke f to sort the input. Suppose that the two algorithms are equivalent on inputs of all sizes smaller than some $\tilde{t} \geq t - 1$. We claim that the two are also equivalent on inputs of size $\tilde{t}+1$. Because $\tilde{t}+1 \geq t$, the algorithm $[(t, f), (\infty, r(\text{CALLER})]$ calls r. The callee, r sorts the input by calling $[(t, f), (\infty, r(\text{CALLER})]$ on subsequences. Since the length of subsequences is at most $\tilde{t}$, the behavior of $[(t, f), (\infty, r(\text{CALLER})]$ on the subsequences is the same as the behavior of $r'_{\text{threshold}=t}(f)$. On the other hand, the algorithm $r'_{\text{threshold}=t}(f)$ sorts the length $\tilde{t} + 1$ sequence by running r and calling itself on subsequences. Hence, the two are equivalent.

In fact, in most cases ATSL contains either r or r', but not both. Most of the composed algorithms in ATSL are of the form $r'_{\text{threshold}=t}(f)$, but a few are simple compositions.

Clearly, we could also support other forms of CALLER expressions via equivalences to CALLER-free expressions. For example, we could include in ATSL an algorithm $r''_{\text{threshold1}=t_1,\text{threshold2}=t_2}(f_1, f_2)$ that corresponds to

$$[(t_1, f_1), (t_2, f_2), (\infty, r(\text{CALLER})]$$

for some r. But ATSL cannot construct r'' automatically, and more generally, it cannot generate the code for an arbitrary expression that uses the CALLER notation.

This is a limitation of ATSL, but even with this limitation ATSL can generate codes that correspond to the most frequent use of CALLER-like idioms in hand-coded algorithms.

There are two ways to implement the CALLER notation in a general way, shown in Figure 4.3. One is to pass a function pointer to the caller to each sorting

```
typedef void (*fn)();
typedef void (*sortfn)(datatype*,
              int,
              fn);

void zsort(datatype* A,
           int n,
           sortfn caller)
{
  ...
  caller(A  ,j  );
  caller(A+j,n-j);
  ...
}
```

```
void
  zsort_ss_31_f_inf_zsort_caller
    (datatype* A,
     int n)
{
  ...
  ss_31_f_inf_zsort_caller(A,
                           j);
  ss_31_f_inf_zsort_caller(A+j,
                           n-j);
  ...
}
```

FIGURE 4.3. Two general ways to support the CALLER notation. The two codes show the implementation of an imaginary sorting algorithm, ZSORT, within the expression $[(31, f), (\infty, \text{ZSORT}(\text{CALLER}))]$.

function. This would have some run-time cost and would prevent some interprocedural compiler optimizations. The other way binds the caller at compile time. Suppose that the caller is $[(31, f), (\infty, \text{ZSORT}(\text{CALLER}))]$ and that the function that implements it should be named `ss_31_f_inf_zsort_caller`. We could generate a version of `zsort` that calls the caller by name. That is, the version of `zsort` that we generate invokes `ss_31_f_inf_zsort_caller` to sort sub-arrays. We can generate the caller after we generate this specialized version of `zsort`. All that is required when we generate `zsort` is the full name of the caller. Clearly, the specialized version of `zsort` must have a special name, and in general it cannot be used in other sorting algorithms that use ZSORT(CALLER).

4.4. **Limitations II: Tree Structures.** ATSL only generates codes for expression trees with certain structures, shown in Figure 4.4. This limitation was put in place mostly to structure and limit the size of the search space, not because the code-generation mechanism itself is limited. We explain the limitation assuming that all CALLER sub-expressions have already been transformed into CALLER-free parametrized equivalents.

Admissible trees have depth 4 or less. The root (level 0) is always a size selector. The thresholds of this selector are specified when ATSL is configured.

The children of the root are either leaf algorithms or divide-and-conquer compositions. The algorithms that divide-and-conquer compositions call are either leaf algorithms, or size selectors. Level-2 size selectors always call leaf algorithms.

Divide-and-conquer algorithms and leaf algorithms at any level can be parametrized.

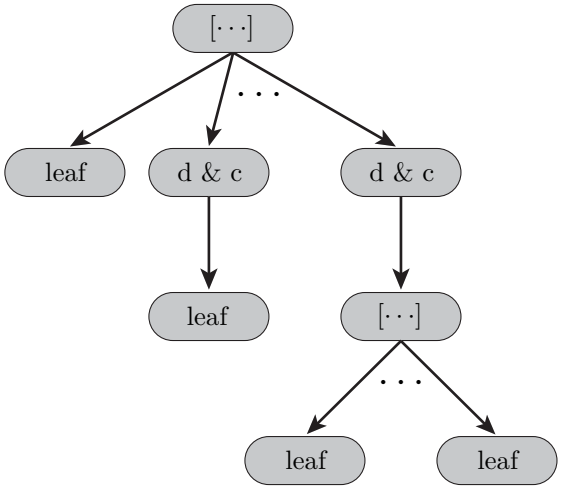

FIGURE 4.4. Possible tree structures for the final algorithm that ATSL produces. The root is a size selector, and its children are either leaf algorithms or divide-and-conquer algorithms (denoted "d & c"). The divide-and-conquer algorithms can be composed with either leaves or with size selectors that call leaves.

Many of the divide-and-conquer algorithms, which can only be present at level 1, are equivalents of CALLER expressions. Level-2 size selectors use the same thresholds as the root, except that if they are called by a CALLER equivalent $r'_{\text{threshold}=t}(f)$, no thresholds larger than t are used.

4.5. **The Repertoire of Building Blocks.** Tables 1 and 2 list the sorting algorithms that ATSL currently uses as building blocks. Table 1 lists leaf algorithm, and Table 2 lists divide-and-conquer compositions. Many of these building blocks are based on existing publicly-available codes.

4.6. **The Generation Schedule.** ATSL's code-generation schedule is driven by the search algorithm, which we describe below, not by large expression trees. Code generation starts at leaf algorithms. Leaf algorithms that are not parametrized require no generation; they are hard coded. From ATSL's view point, these algorithms are not compositions, but they may well be implemented using recursion. For example, ATSL uses recursive variants of quicksort as hard-coded leaf components. The recursive call within such a component is always to a sorting function within the same component, which might be the quick-sort caller or an insertion sort. But ATSL is oblivious to the internal implementation of such components.

Leaf algorithms that are parametrized are the first algorithms that ATSL generates. The possible values for each parameter are listed in a configuration file.

Next, ATSL generates all the possible compositions of divide-and-conquer algorithms with leaf algorithms. These compositions are then used for certain

Type	Source	Variant name in the original source
quicksort	1	*recursive*
quicksort	1	*pure-iterative*
quicksort	1,*	*iterative*
quicksort	2	*Hoare*
quicksort	2	*Lom*
quicksort	4	calls insersionsort on small sub-arrays
mergesort	1	*recursive*
mergesort	1	*optimized-iterative*
mergesort	2	
heapsort	2	
heapsort	3	Originally an internal subroutine
radixsort	3	*LSDRadixSort*
radixsort	3,*	Automatically-generated versions of *LSDRadixSort*
insersionsort	2	
insertionsort	4	Originally an internal subroutine
bubblesort	2	*BubbleSort*
bubblesort	2	*BubbleSortBi*
shellsort	2	
selectionsort	2	

List of sources for the implementations:
1. Implementations by LaMarca and Ladner [14].
2. Publicly-available code by Benedikt Meurer.
3. Publicly-available code by D. J. Bernstein, partially based on [22] and [7].
4. The STL C++ Library (on the build platform).

*. Parametrization by code generation.

TABLE 1. The leaf algorithms that ATSL uses. The table shows the algorithm type, its origin, and how the particular variant was called in the original sources.

performance measurements, which are part of the search algorithm. Because these compositions are only used for performance measurements within the search algorithms, the composition is performed using function pointers to the leaf algorithms, not using the macro-instantiation technique shown in Figure 4.2. This composition method speeds up the build process. These compositions are temporary and are only used as callers of the leaf algorithms whose performance we measure; their own performance is not measured at all. Therefore, the so the slight performance penalty of function pointers *on the caller of the algorithm being benchmarked* is harmless.

The next generation stage builds all the candidate algorithms that are rooted at level 1 of admissible trees. This includes divide-and-conquer compositions with

Type	Source	Variant name in the original source & comments
quicksort	1	*recursive*
quicksort	1	*iterative-base*
quicksort	1	*multipartition*
quicksort	4	
samplesort	5	*SuperScalarSampleSort*
mergesort	1	*Multi-mergesort*
mergesort	6,*	*N Pass*es; a (N+1)-level algorithm
mergesort	—	A multi-way mergesort; our implementation
radixsort	3	*MSDRadixSort*; most-significant bits first
radixsort	3,*	Automatically-generated versions of *MSDRadixSort*

List of sources for the implementations:
1. Implementations by LaMarca and Ladner [14].
2. Publicly-available code by Benedikt Meurer.
3. Publicly-available code by D. J. Bernstein, based on [22] and [7].
4. Microsoft's STL C++ Library.
5. Implementation by Sanders and Winkel [19].
6. Implementation by Wickremesinghe et al [25].
*. Parametrization by code generation.

TABLE 2. The divide-and-conquer algorithms that ATSL uses.

leaves, and divide-and-conquer compositions with size selectors that call leaves. For each parametric variant of a divide-and-conquer algorithm, ATSL builds two compositions, one with a leaf and one with a size selector. The threshold in the size selector is automatically tuned. There is no need to re-generate leaves at this stage; they are all already generated (or they are hard-coded).

Finally, ATSL constructs a single root size selector.

5. Exploring the Search Space

The final sorting code that ATSL generates is a deep composition of many sorting algorithms, some of them parametrized. This section explains how ATSL searches this huge design space of potential algorithms.

5.1. **Factoring Out the Dependence on Input Distributions.** Some sorting algorithms benefit from specific input distributions. For example, insertionsort worst-case running time is $O(n^2)$, but it runs in linear time if the input in sorted or almost sorted. Mergesort, on the other hand, runs in $\Theta(n \log n)$ time no matter what the input is. ATSL searches the design space by direct running-time measurements of sorting algorithm. Clearly, at least for some algorithms the running-times depend on the inputs.

Our design goal has been to address this issue by avoiding algorithms that are very slow on specific inputs (e.g., random, with repetitions, repetitions in the

low or high bits) while favoring algorithms that benefit from regularities (e.g., repetitions). That is, we would like the worst-case behavior of the resulting algorithm to be close to optimal. On the other hand, we are willing to accept slightly sub-optimal worst-case behavior in return for significant speedups on more structured inputs.

We achieve this design goal by generating inputs that mix several distributions of permutations. In every input that ATSL generates, every element in position i, where $i \bmod 4 = 0$, is an independent uniform random sample from the integers between 1 and some maximal value ($2^{31} - 1$ for 32-bit signed integers, etc.). Elements such that $i \bmod 4 = 1$ have their upper bits set to a fixed random value (half the bits), but the lower bits are random and uniform. Elements such that $i \bmod 4 = 2$ have their lower bits fixed (a quarter of the bits) and the upper bits are random and uniform. Finally, elements such that $i \bmod 4 = 3$ can have only two values; the two possible values are random and the choice between them is random with probability 1/2.

This method acheives our design goal in that is is likely to expose algorithms that perform poorly on random inputs and algorithms that perform well only when many of the bits of the keys are random. On the other hands, algorithms that benefit from repetitions are likely to be favored by ATSL, because 12.5% of the keys all have a single value and another 12.5% of the keys have a different fixed value.

5.2. **Running-Time Vectors.** The object that we use to represent the performance of a given sorting algorithm in a given context is a *running-time vector*. A running-time vector is a vector of normalized estimated running times $[t_1, t_2, \ldots, t_k]$ for that algorithm to sort arrays of certain sizes $[n_1, n_2, \ldots, n_k]$. The sizes are determined by the configuration of ATSL. We currently use the sizes

$$2^4, 2^5, \ldots 2^{10}, 2^{12}, 2^{14}, 2^{24}, 2^{25}, \ldots, 2^{27} ,$$

along with four large non-power-of-2 sizes between 2^{18} and 2^{20}; some of the powers of 2 are a factor of 2 apart and some are a factor of 4 apart. The sizes do not need to be powers of 2: any set of positive integer sizes can be used, except that ATSL ignores sizes for which the array size is larger than main memory.

For each size, a running-time vector stores an estimate running time per array element. For example, a running-time vector $(1.03 \times 10^{-7}, 2.32 \times 10^{-7}, \ldots)$ means that we expect the algorithm to take $2^4 \times (1.03 \times 10^{-7})$ seconds to sort arrays of size 16, $2^5 \times (2.32 \times 10^{-7})$ seconds to sort arrays of size 32, and so on. (In our implementation, the units are not seconds, but this is irrelevant to the description of the system.)

We allow ∞ in running-time vectors. An infinity means that the estimated running time is too long to be relevant to ATSL. The inclusion of infinities allows us to avoid very long test runs, for example of $\Theta(n^2)$ algorithms. When we measure running times to create running-time vectors, we use four thresholds,

c, n_l, n_s and t_l to assign ∞'s. The parameter t_l is the measured running time of a fixed algorithm F (specifically, the one from the STL library) on arrays of size n_l, where n_l is chosen to be larger than the processor's caches. If during the generation of a running-time vector we measure a per-key running time t on an array of size $n \geq n_s$, and if the total running time T satisfies

$$T = tn > c\frac{t_l}{\log_2 n_l} n \log_2 n$$

then we assign ∞ to all the running-time-vectors for sizes larger than n. This rule reflects the following assumptions:

- The running time per key of sorting algorithms is monotonic for all $n \geq n_s$. The worst-case number of operations is monotonic for all the deterministic sorting algorithms that we are aware of, and the worst-case (over the inputs) expected number of operations in randomized algorithms is monotonic as well. Nonmonotoniciries can occur due to specific inputs (e.g., a small random input versus a large but already sorted input), and due to random choices that randomized algorithms make. However, we prune an algorithm using this rule because of a nonmonotonicity when it runs slowly on small inputs. Even if the same algorithm performs better on some larger inputs, the prunning decision is reasonable: it prunes an algorithm that is at least sometimes much slower than the fixed algorithm.
- The running time of the fixed algorithm F for $n \geq n_s$ is at most $(t_l/\log_2 n_l)n \log_2 n$. This assumption is valid when the fixed algorithm is a $\Theta(n \log n)$ algorithm, such as mergesort or heapsort, and when an array of size n_l is significantly larger than the largest cache. At small array sizes, there are fewer cache misses, so even if the number of machine instructions does behave like $\delta n \log_2 n$ for some δ, the the running time grows faster than $\Theta(n \log n)$ due to an increasing cache-miss rates. But for large array sizes, the assumption is valid.
- Running times that are more than a factor of c larger than that of the fixed algorithm are irrelevant. This assumption is valid because ATSL can always choose to use the fixed algorithm F rather than a much slower competitor.

Under these assumptions, the pruning rule is correct. It is based on the trivial observation that if the running-time of F at n_l is $T_{F,n_l} = \gamma n_l \log_2 n_l$ for some γ, then $\gamma = t_l/\log_2 n_l$.

In our implementation, F is the platform's STL sorting algorithm, n_l is such that the array is 8 times larger than the level-2 cache, $n_s = 1024$, and $c = 5$. The choice of c is fairly arbitrary. Larger c's lead to less pruning and a more accurate estimation of running-time vectors, at the expense of more work in the search algorithm.

To estimate running-times on small running times without relying on high-resolution timers, we duplicate the input array several times. The duplication is

done so as to factor out cache effects. We allocate two arrays D and J that are twice as large as the level-2 cache, to ensure that each call to the sorting code sorts a subarray whose elements are *not* in the level-1 or in the level-2 cache. To measure the running time of an algorithm on a small array A of size n, we place copies of A in D, contiguously, until the remaining space is smaller than A. We then read sequentially all the elements of J, to evict any elements of D from the cache. The sequential access is based on the assumption that here that the cache does not detect streaming; if it does, we could read J in an apparently random order. Now we sort all the copies of A that we placed in D, measure the total running time, and divide by the number of copies. This causes the copies of A to be sorted with a mostly cold cache (except perhaps for a single partial cache line that was read into the cache when the previous copy was sorted). The effect of a cold cache is approximately the same on all sorting algorithms: they incur compulsory cache misses to read the input into the cache.

5.3. **Generating Context-Independent Running-Time Vectors for Leaf Algorithms.** The first phase in the exploration of the search space is straightforward. ATSL generates all the parametrizations of leaf algorithms, and then generates a running-time vector for each one, by measuring directly its running time on generated random inputs. As explained above, the estimates on small inputs are measured by duplicating the input and reflect cold caches.

To make the estimates reliable, we average 15 runs per algorithm per size on large sizes (2^{14} elements and larger) and 50 runs on small sizes.

5.4. **Generating Context-Dependent Running-Time Vectors for Leaf Algorithms.** Consider a CALLER-equivalent $r_{\text{treshold}=t}(f)$ that calls a leaf f on small sub-arrays. On certain input distributions and array sizes, the sub-arrays that f sorts are highly biased, and the bias depends on r.

For example, consider inputs of the form $(0, -1, 1, -1, 1, -1, 1, \ldots)$. Let f be insertionsort, let r_q be quicksort variant that uses the exact median as a pivot, and let r_m be mergesort. Assume that both r_q and r_m call f on sub-arrays smaller than some value n_0. The sub-arrays that f receives from r_q are likely to contain only one value, so f will run on them in linear time. But the sub-arrays that f receives from r_m are likely to contain a random mixture of the 2 possible input values, leading to quadratic running times for f.

The inputs that ATSL currently uses do not appear to generate such strong biases, but we have designed ATSL so that future versions will allow the user to plug in an input generator, to tailor the resulting sorting subroutine that ATSL produces to specific input distributions. As shown above, for some inputs, the sub-arrays can be strongly biased.

Therefore, we measure each leaf algorithm in the context of each divide-and-conquer algorithm. We generate all the divide-and-conquer variants (under all possible parametrizations, including CALLER-equivalent thresholds). We run each such variant r with each leaf f, and measure the running times for f in the context

of the particular divide-and-conquer variant. We ignore the running time of the composed algorithm now; we only use the composition to create a context for f.

When r generates inputs for f during its normal operation, the input sizes to f are not necessarily the input sizes in our running-time vectors. To make running-time vectors comparable, we assign all the per-key running measurements to nominal sizes. For example, all running times on arrays with 1–24 elements will be assigned to size 16, running times with 25–48 will be assigned to size 32, and so on. This is why the running-time vectors store per-key times rather than total times.

Also, in general r does not generate enough input sizes to estimate an entire running-time vector for f. For example, recursive mergesort with threshold 256 might generate only sub-arrays of one size, if the total input size is a power of 256. Even with a random total input size, mergesort might only generate inputs with size between 129 and 256. In this particular case, the performance of f on other inputs is irrelevant, but in other cases there might be rare but possible input sizes. To address this issue, we complete context-dependent running-time vectors by copying missing estimates from the context-independent vector of f.

5.5. **Generating Size Selectors.** We now explain how ATSL constructs size selectors. The construction of a size selector depends on a *candidate list*. This is a list of sorting algorithms $a_1, a_2, \ldots a_\ell$, each with an associated running-time vector. We denote the running time vector associate with a_j by t^{a_j}. The size selectors that ATSL generates are always of the form $[(n_1, s_1), (n_2, s_2), \ldots, (n_m, s_m)]$, where the sequence $n_1, n_2, \ldots, n_m$ is a prefix of the sequence of sizes $n_1, n_2, \ldots, n_k$ that characterizes run-time vectors.

We construct a selector s from a candidate list by assigning to s_i the candidate algorithm whose associated running-time vector at n_i is minimal. That is, $s_i = a_j$ where

$$j = \arg\min \{t_i^{a_1}, t_i^{a_2}, \ldots, t_i^{a_\ell}\} \ .$$

After ATSL estimates all the context-dependent running time vectors for a divide-and-conquer variant r, it creates a size-selector s for r. That is, we construct a size selector s that will perform nearly optimally in r. We denote the resulting size selector by s_r. (If r composes two or more sorting algorithms, as in sample sort, we construct a size selector for each sorting-algorithm argument of r.) In the construction of the size selector s_r, the candidate list includes all the leaf algorithms under all parametrizations, and the running time vectors associated with them are *the context-dependent running time vectors, where the context is r.*

The last step on the synthesis of the final algorithm constructs the level-0 size selector s_T. Its candidate list include all the parametrized leaf algorithms and two instantiations of each parametrized divide-and-conquer algorithm r. One instance of a divide-and-conquer variant r is $r(s_r)$, where s_r is the nearly-optimal size selector that was constructed for r. The other variant is $r(f)$, where f is a

leaf algorithm whose r-dependent running-time vector has a small 1-norm (sum of elements). We always select f for r from the leaves that populate s_r.

Before we can construct s_T, we need to estimate the running-time vectors for each $r(s_r)$ and $r(f)$ in the candidate list. So far, we only constructed running-time vectors for leaf variants. We run these algorithms and measure their running times to construct the running-times vectors. Now each algorithm in the candidate list is associated with a (context-independent) running-time vector, so we can construct the selector s_T.

5.6. **The Overall Search Algorithm.** We now summarize the overall search algorithm. We start by instantiating each leaf variant and measuring its performance to construct a context-independent running-time vector for it. Then we instantiate each divide-and-conquer variant r as a context, and measure the performance of each leaf variant in that context. We fill missing values in a context-dependent running-time vector with values from the corresponding context-independent vector.

Once we have all the r-dependent vectors, we construct the size selector s_r for r, and we find a good leaf variant f for use in r without a size selector. We measure the running times of $r(s_r)$ and $r(f)$ and add them to the candidate list of s_T, the top-level selector. The final step is the construction of s_T.

6. Software Engineering

6.1. **Design Goals.** This chapter highlights several software-engineering issues in the design of ATSL. The engineering goals that we tried to achieve in the design are:

- Portability. We tried to ensure that ATSL can run on a wide variety of platforms. Portability is important for two reasons. First, it enhances the practical value of the system. Second, it allows us to evaluate it on multiple platforms. We viewed this second issue as critical, because without evaluating ATSL on multiple platforms it would not have been possible to validate our claims regarding the value of automatic generation of sorting algorithms.
- Ease of integration of new algorithms and implementations. Novel algorithmic and implementation ideas for sorting algorithms are invented all the time. It is important to keep ATSL open ended so that new variants can be incorporated into the library easily.
- The resulting sorting code should be callable from C. C-callable functions can be called not only from C programs, but also from C++ programs, Java programs, Fortran programs, and many other environments. Therefore, ensuring that the resulting code is callable from C allows it to be used in many programs. However, a C++ compiler is required in order to build ATSL itself and in order to compile the resulting code. ATSL uses

variants that are implemented in both C and C++ (in principle, assembler and Fortran variants should also be easy to incorporate), so calling programs should be linked with both the C and C++ standard libraries.

6.2. **Design.** We now describe the overall design of ATSL. The description focuses on the design of the code-generation and tuning phases. The final sorting code that ATSL produces consists of only building-block instantiations, with no ATSL code whatsoever. Therefore, the interesting parts of the design all belong to the code-generation and tuning phases.

ATSL is implemented in C++ and its design is based on three kinds of objects. To present these objects, we use a small set of building blocks,

- $\text{SAMPLESORT}_{\text{sample-size}=p_1}(f_1, f_2)$,
- INSERTIONSORT, and
- $\text{RADIXSORT}_{\text{radix}=p_2}$.

The expressions p_1 and p_2 denote numeric integer parameters; $f_1$and f_2 denote sorting-algorithm parameters. The three kinds of ATSL objects are:

Parameter-type objects: These objects represent the type of a parameter. In the example, we have three types of parameters: the sample-size type, which characterizes p_1 in SAMPLESORT, the radix type, which is used in RADIXSORT, and the sorting-algorithm type, which characterize f_1 and f_2. Numeric types, like radix, have a set of possible values. Multiple algorithms can use a single parameter type. For example, the CALLER-threshold type is used by most divide-and-conquer algorithms. These objects essentially provide a single service that returns an iterator for the possible values of this parameter.

Building-block objects: These represent our building blocks, such as INSERTIONSORT and $\text{RADIXSORT}_{\text{radix}=p_2}$. For parametric and composed algorithms, the representation includes the types of their parameters. These objects provide the following services:

Parameter enumeration. This service allows ATSL to determine the number and types of parameters to this building block. For example, for SAMPLESORT this service reports 3 parameters, of types sample-size, sorting-algorithm, and sorting-algorithm.

Code generation. A building-block object can instantiate a parametric, divide-and-conquer, or selector algorithm. The code-generation service returns two items: the program text of the fully-instantiated algorithm, say

$$\text{SAMPLESORT}_{\text{sample-size}=7}(\text{INSERTIONSORT}, \text{INSERTIONSORT}) \ ,$$

and an object representing this instantiation. This object is described below. There is no way to partially instantiate a building block, say to

construct

$$\text{SAMPLESORT}_{\text{sample-size}=7}(\text{RADIXSORT}_{\text{radix}=p_2}, \text{INSERTIONSORT}) ,$$

where p_2 is unbound.

Run-time instantiation and execution. The object can also execute a fully-instantiated parametrization without generating and compiling code. That is, ATSL can request that SAMPLESORT be executed with parameters

$$\begin{aligned} p_1 &= 7 \\ f_1 &= \text{RADIXSORT}_{\text{radix}=16} \\ f_2 &= \text{INSERTIONSORT} . \end{aligned}$$

Obviously, the type of the arguments f_1 and f_2 is such that the name of the corresponding sorting function can be determined. This service is used in the computation of context-dependent running-time vectors (in this case, for $\text{RADIXSORT}_{\text{radix}=16}$ and for INSERTIONSORT).

Fully-instantiated-algorithm objects: These represent fully-instantiated algorithms, such as

$$\text{SAMPLESORT}_{\text{sample-size}=7}(\text{INSERTIONSORT}, \text{INSERTIONSORT}) .$$

Note that a non-parametric building block like INSERTIONSORT is represented by both a building-block object and by a fully-instantiated object. The objects provide these services:

Naming. The object can return a string with the name of the C-callable sorting function that it represents. This is used by building-block objects to generate code for compositions.

Execution. The object can run the C-callable function. ATSL invokes this service when it computes the context-independent running-time vector for the algorithm.

The core of ATSL uses these objects to explore the search space and to find an efficient algorithm. ATSL works bottom-up on the set of admissible trees. It maintains two sets of sorting algorithms: a static set of building block objects, and a growing set of fully-instantiated-algorithm objects. At each phase, ATSL enumerates the building blocks. For each building block, it iterates over all of its parameter values. If some of the parameters are sorting-algorithms, it generates their context-dependent running-time vectors (using the runtime instantiation and execution service of the building block). It then decides on the compositions that should be generated, and generates them for use as fully-instantiated algorithms in the next higher level of the tree.

When ATSL enumerates building blocks, the enumeration is restricted, in order to construct only admissible trees. For example, at level 1, the enumeration excludes size selectors, because we do not allow a size selector to call another size selector.

The enumeration of fully-instantiated algorithms for use as composition parameters is sometimes restricted, too. For example, the only fully-instantiated algorithms that a divide-conquer algorithm is allowed to call are a size selector and one of the leaves that the size selector calls.

6.3. **Implementation.** The implementation of ATSL consists of static C++ code, automatically-generated C++ code, and scripts. The partitioning into both C++ programs and scripts was driven by the need to invoke the compiler to generate both sorting algorithms and the classes that represent them. To make the system portable, the scripts are generated by a configuration program that runs at the beginning of the build process.

A run of ATSL is initiated by calling a small and simple script. The script compiles and runs a configuration program whose role is to produce the scripts that will run the rest of ATSL. The output of the configuration program depends on several parameters that it receives from the user, such as the command that invokes the compiler and so on.

Because the design of ATSL evolved during its development, its C++ class hierarchy does not correspond exactly to the abstract design that we described in Chapter 6.2. The differences are not fundamental and can be corrected by refactoring the code.

The representation of building blocks is perhaps the most interesting aspect of the implementation. We describe this aspect in detail, because the details show how additional building blocks can be added to the system.

The representation of a building block usually consists of three parts. The main part is the class whose single object will represent the building block. In our implementation, each building block uses a separate class with a hand-written static source code. Typically, the code generation service of this class uses either a C macro or a C++ template. This macro or template forms the second part of the implementation. Chapter 4 explained this technique. Using the example of Chapter 4 and assuming that the macro is defined in the file `quicksort_medianof.h`, the code generation for a particular instantiation produces the code

```
#include "quicksort_median.h"
QUICKSORT_MEDIANOF_M_F(3,insertionsort)
```

(the string `insertionsort` is obtained from the fully-instantiated-algorithm object that represents this algorithm). That is, the actual implementation of a composed or parametrized algorithm r is a macro, and its parameters are macro parameters. In a composed algorithm, the algorithms that it calls are also macro parameters. This allows the compiler the greatest opportunity to optimize the resulting codes, as opposed to passing parameters as function arguments. The implementation of a composed-algorithm building block also includes a *function* (as opposed to a macro) that retrieves the building-block's parameters from global variables. This function is called by the runtime-instantiation service. The third part of the implementation is the definition of parameter types specific to this

building block. Currently, all the parameter types are defined in one configuration file.

Not all the building blocks are implemented using macros or templates. Some of ATSL's building-block objects use custom C++ code generators to generate parametrized algorithms. At the other extreme are non-parametrized algorithms whose code generators simply return static source code from a file.

To add a new building block to ATSL, the developer implements a class representing the building block, including the code generator, *and registers this class* in ATSL's static set of building blocks.

ATSL runs under Windows, Linux, and MacOS X, and has been tested with GCC (on all platforms), Intel's C++ compiler (under both Windows and Linux) and Microsoft's Visual C++ compiler (under Windows).

A run of ATSL generates a sorting library for a particular data type and a particular partial order on that data type. The ability to generate libraries for sorting arrays of primitive data types (`int`, `double`, etc) is built into ATSL. To sort structures, the user must provide ATSL with the declaration of the data type, using a `typedef` statement, and with the name of field on which the structures are sorted.

7. Experimental Results

We present results from runs of ATSL on six different computers, three different compilers (on a single computer), and on several input distributions. The experiments attempt to answer several questions:

- Are ATSL's resulting algorithms consistently and significantly faster than hand-coded algorithms?
- Do ATSL's resulting algorithms adapt effectively to the computer's architecture?
- Do ATSL's resulting algorithms adapt to different optimizing compilers?
- Do ATSL's resulting algorithms perform well on input distributions other than the one with which they were selected?

7.1. Experimental Platforms.

We used the following computers for the experiments:

3.2GHz Pentium 4: The computer has 2GB of RAM, a processor with a 1MB level-2 cache, and runs Linux, kernel version 2.6.11. We used the Intel C++ computer version 8.0 on this computer.

3.0GHz Pentium 4: The computer has 2GB of RAM, a processor with a 1MB level-2 cache, and runs Linux, kernel version 2.6.11. We used GCC version 3.3.5 on this computer.

0.5GHz Pentium 3: This computer has two CPU's (we only used one) with 512KB level-2 caches, 256MB of RAM, and runs Linux, kernel version 2.6.11. We used GCC version 3.3.5 on this computer.

	3.2 GHz	3.0 GHz	1.6 GHz	0.7 GHz	0.5 GHz
Unsigned integer	13	16	9	24	27
Structure	18	11	7	7	15

TABLE 3. ATSL build times in hours on several platforms, denoted by clock speed. See the text for further descriptions of the machines.

1.6GHz AMD Opteron: This computer has two CPU's (we only used one) with 1MB level-2 caches, 8GB of RAM, and runs a 64-bit version of Linux, kernel version 2.6.8. We used GCC version 3.3.5 on this computer.

0.7GHz PowerPC G4: This computer has 256MB of RAM, 256KB level-2 cache, and runs MacOS X version 10.2.8. We used GCC version 3.1 on this computer.

2.8GHz Pentium 4: This computer has 512MB of RAM, a processor with a 512KB level-2 cache, and runs Windows XP. On this computer, we used three compilers, GCC version 3.2, Microsoft's C++ compiler version 7, and Intel's C++ compiler version 8. We used this computer mostly to compare ATSL under different compilers.

This collection of test platforms contains several processor architectures, two by Intel, one by AMD, and one by Motorola and IBM, and three different compilers.

7.2. **Build Times.** In our experiements, the running time of ATSL itself ranges from several hours to 24 hours. Running times of up to about 16 hours allow for overnight builds, which are reasonable for the end user. Running time over 16 hours require either a weekend build or avoiding the use of the machine during working hours. The two drawbacks of long running times are the effects of build failures and the difficulty of debugging and developing ATSL itself. For the end user, the drawback is that the failure of an overnight build usually takes 27 hours to detect and correct. A build might fail if the user supplies invalid configuration parameters, such as compiler flags. Most build failures occur early in the build and can be quickly corrected, but a late failure can waste significant time before it is detected and corrected. For the ATSL developer (us), the drawback is that the effect of small changes in ATSL on the performance of the resulting algorithms usually takes at least 24 hours to evaluate. To fully assess the effect of modifications, the developer usually runs ATSL on several platforms and datatypes, which increases the evaluation time even further.

Table 3 describes some sample build times. The table shows some irregularities, most of which we have not investigated. The differences between the 3.2 and 3.0 GHz Pentium 4 runs may be due to different compilers: GCC on the 3.0 GHz machine, and Intel's compiler on the 3.2 GHz machine.

7.3. **Results.** The first sets of results compare the performance of several hand-coded algorithms to the performance of ATSL's generated algorithm. The results

are shown in in Figures 7.1, 7.2, 7.3 and 7.4. In these graphs, the performance of hand-coded algorithms is shown using their running-time vectors, but the performance of ATSL's algorithm is shown on additional random sizes.

All the running times were measured on ATSL-style inputs, described in Chapter 5. This may seem like cheating, but the results still contain useful information. First and most importantly, we also show experiments on other distributions later. Second, any specific choice of input distributions may hide some algorithmic behaviors. Third, the results on ATSL-style inputs does show whether the search and synthesis whether or not the search algorithm is effective, ignoring the fact that the performance of a particular algorithm in this space also depends on the input distribution.

The most obvious fact that emerges from these graphs is that the performance of the hand-coded algorithms varies greatly, even though most of them are carefully designed and implemented, and many are optimized to exploit data caches.

Another fact that emerges from the integer-sorting results is that on all the machines, a LaMarca-Ladner $O(n \log n)$ algorithms is fastest among the hand-coded variants on arrays of up to several hundred elements, and radix sort is fastest on larger arrays.

The algorithms that ATSL constructs follows this pattern: they use an $O(n \log n)$ variant on small arrays and a radix sort variant on large arrays. They are almost always faster than the hand-coded algorithms. ATSL's algorithms are usually not significantly faster thna all the hand-coded algorithm, but from some of them. This statement is not as trivial as it may seem, considering that most of the hand-coded algorithms were shown to be fast by their inventors or developers to be fast. The superiority of ATSL's algorithms over the hand-coded ones is due to two facts: the fact that ATSL measures the different codes in the correct context and selects the best, and the fact that ATSL synthesizes algorithms that are often more complex than the hand-coded ones. The second advantage is especially significant on radix-sort variants.

The behavior of the algorithms when sorting structures is similar, except that ATSL does not use radixsort variants on floating-point keys. But we again see the high-performance of the LaMarca-Ladner algorithms and we again see that ATSL's algorithm beats them.

The next set of results compare the performance of ATSL's generated algorithm on five different distributions, the one with which ATSL generates running-time vectors and four others. The other distributions were a random distribution, ascending and descending inputs with the keys $0, 1, \ldots, n$, and inputs in which elements were selected randomly and uniformly from a set $\{r_1, r_2\}$ of two random numbers. These results are shown in Figure 7.5 and 7.6.

When sorting integers, ATSL's algorithm performed well on all the input distributions except ascending and descending inputs. The poor performance of ATSL's algorithm on these inputs is not due to their permutation, but due to the fact that all the keys lie in a small interval. Such inputs are not handled well

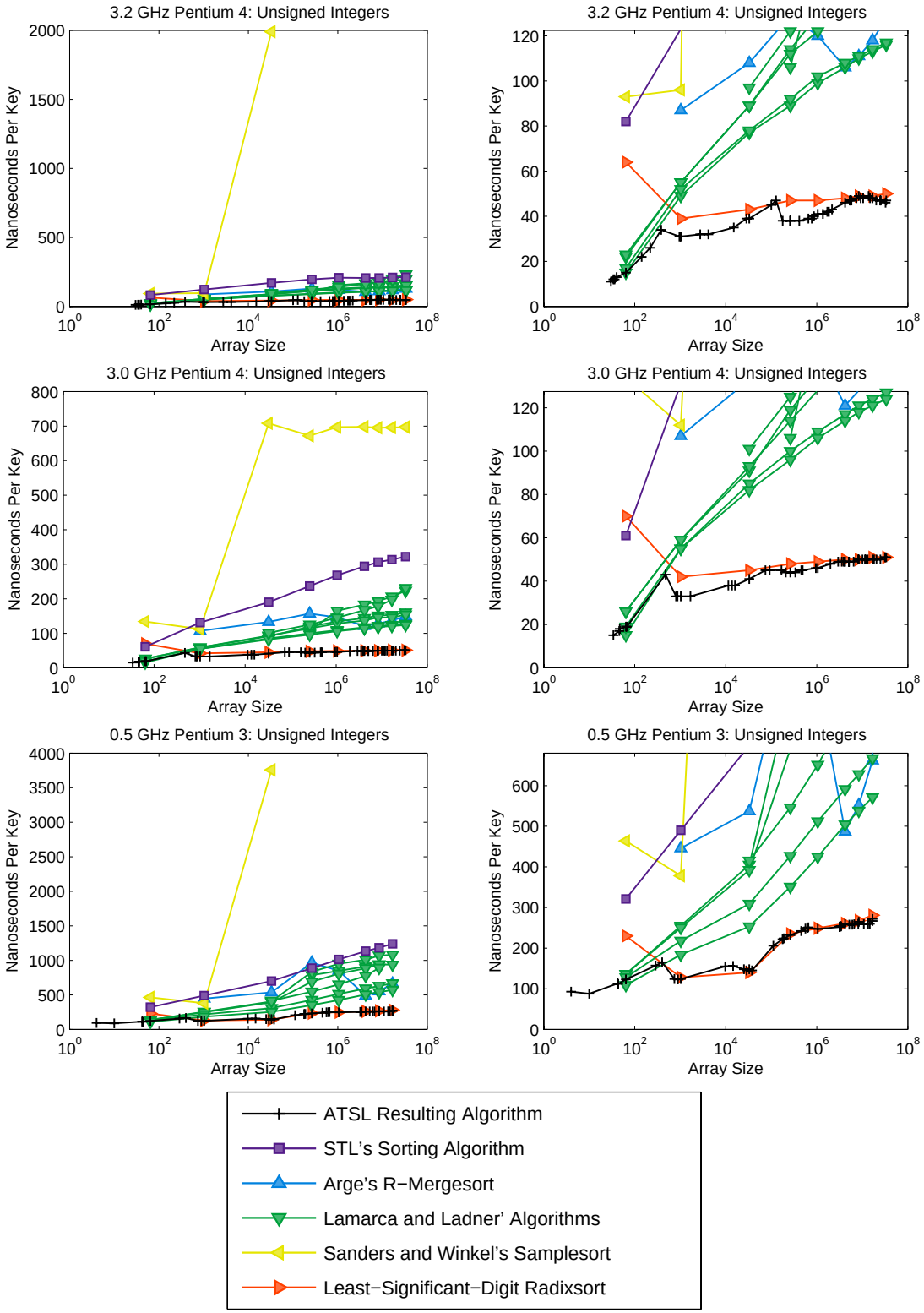

FIGURE 7.1. The performance of hand-coded sorting algorithms versus ATSL's generated algorithm on 3 different computers (results on additional computers are shown in Figure 7.2). The graphs on the left show all the data points in the running-time vectors; the graphs on the right only show the shorter running times.

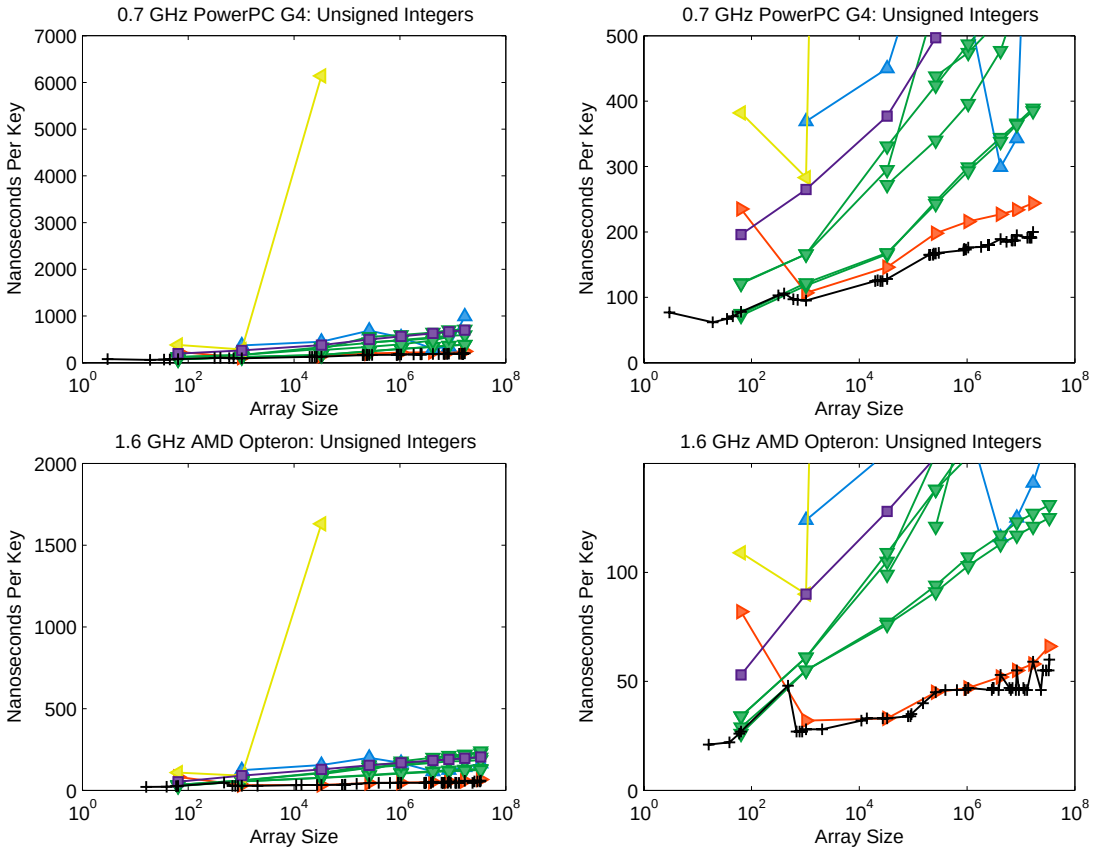

FIGURE 7.2. Performance of hand-coded algorithms versus ATSL's algorithm. Continued from Figure 7.2.

by radixsort, which is usually used by ATSL's algorithm to sort large arrays of integers.

When sorting structures the behavior of ATSL's algorithm is more consistent, primarily because the floating-point keys caused ATSL to exclude radixsort. The performance of the algorithm on random inputs is slightly poorer than the performance on ATSL's training distributions, which is expected, and the performance on other distributions is better than predicted.

Figure 7.7 shows, quantitatively, that the algorithms that ATSL produces on different machines differ significantly from one another. We took the algorithm that ATSL produced on a 0.7 GHz G4 processor and measured its performance on a 0.5 GHz Pentium 3 processor. We used the GCC compilers on both machines. On ATSL's usual distribution, the native algorithm produced on the Pentium is faster, sometimes by almost 35%. This is what we expect. If this was not the case, then ATSL failed to select a good algorithm on the Pentium; but it did not fail. On distributions consisting of ascending and descending values, the G4 is sometimes much faster than the native algorithm and sometimes much slower.

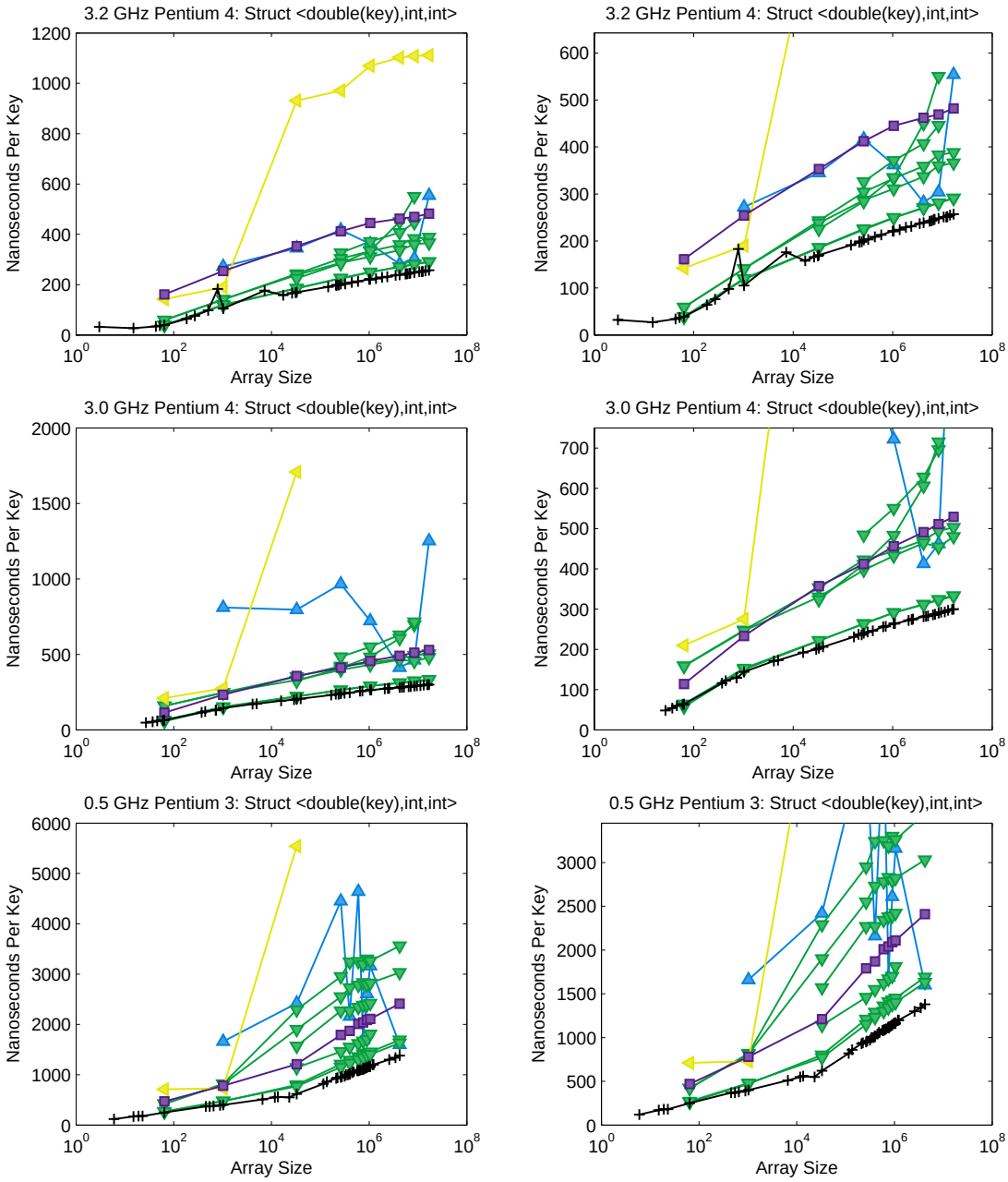

FIGURE 7.3. Performance hand-coded and ATSL's algorithm on structures containing a `double` and two `ints`; the `double` is the sorting key; results on additional computers are shown in Figure7.4).

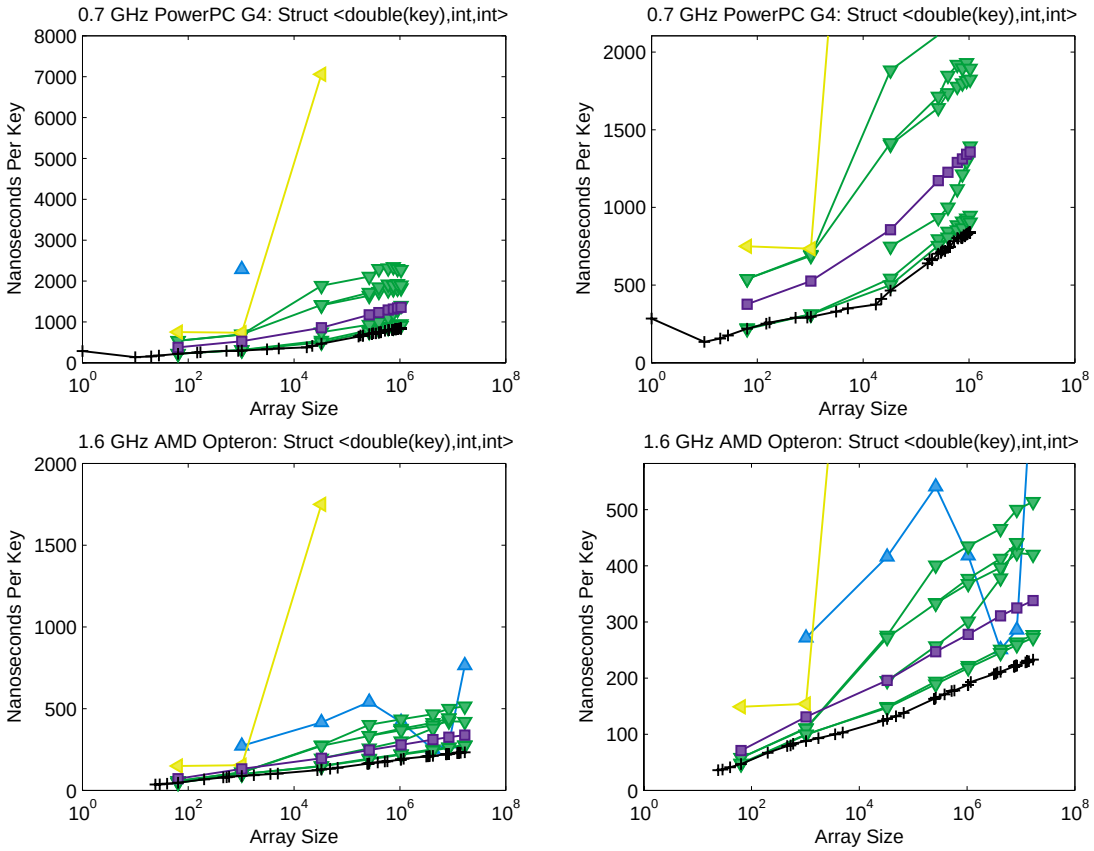

FIGURE 7.4. Performance of hand-coded algorithms versus ATSL's algorithm. Continued from Figure 7.3.

The main implication of this is that the algorithms that ATSL selected on the Pentium and on the G4 have very different behaviors. This is a result of the architectural adaptivity of ATSL.

Figure 7.8 shows that ATSL adapts to specific compilers, not only to specific architectures. Running ATSL on the same machine with 3 different compilers produced results that are different not only quantitatively, but qualitatively. The primary reason for this is the different optimizations that the compilers apply. The existence or lack of a specific optimization can change the relative speed of two sorting algorithms.

ATSL indeed generates different algorithms under different compilers. Under all compilers, ATSL used variants of most-significant-digit radixsort for intermediate input sizes and variants of least-significant-digit radixsort for large sizes. The specific variants and the specific input ranges, however, were different under different compilers. On small array sizes, the differences between the algorithms were even greater. Under Intel's compiler and under GCC, ATSL selected a version

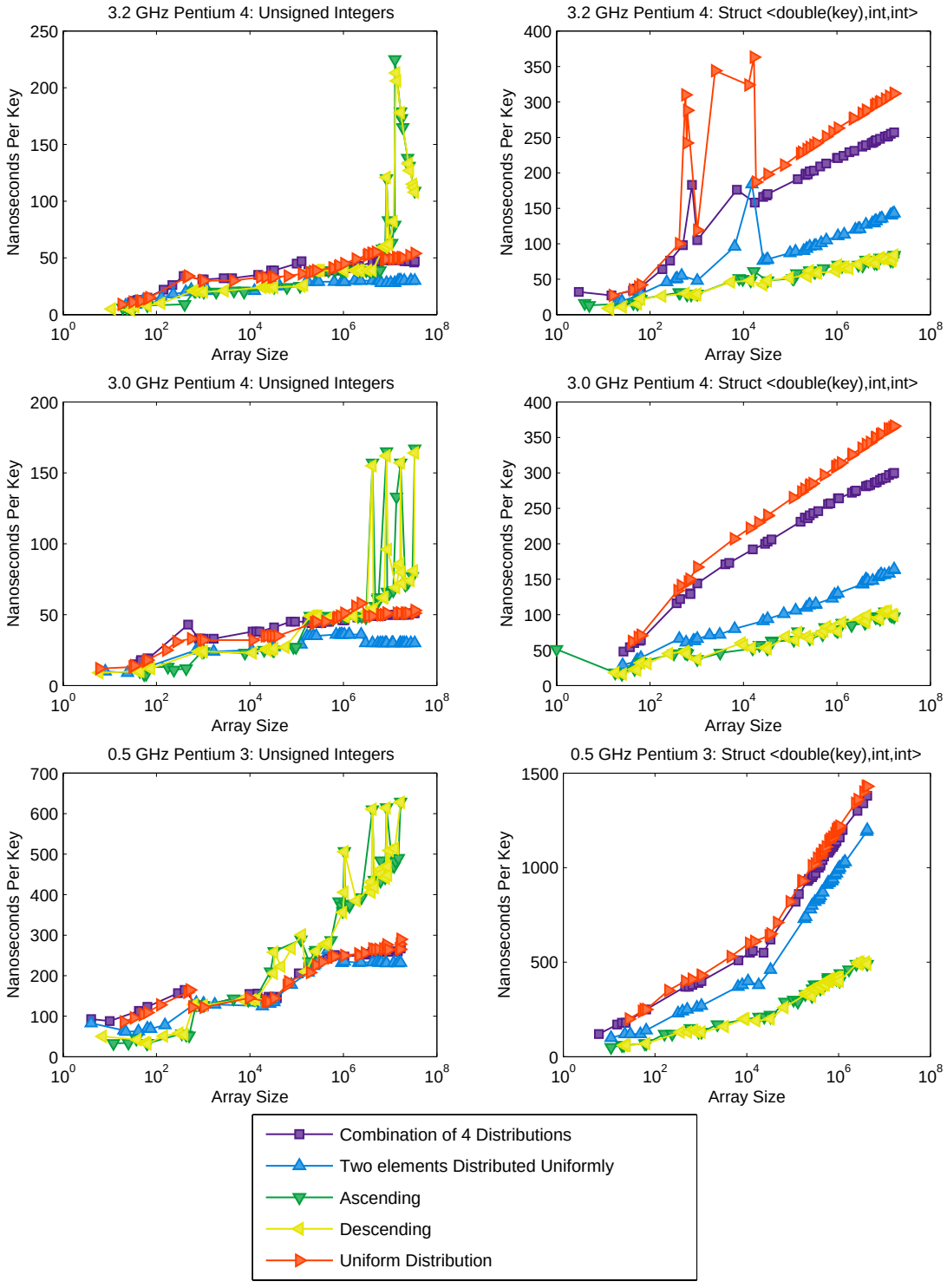

FIGURE 7.5. The performance of ATSL's generated algorithm on five distributions, the one with which it was selected and four other distributions. Continued on Figure 7.6. The erratic behavior on the upper right graph is probably due to an unexpected load on the benchmark machine. The eratic oscilating behavior on the graphs on the right side of the figure (and also in the graphs shown in Figure 7.6) is explained in the text.

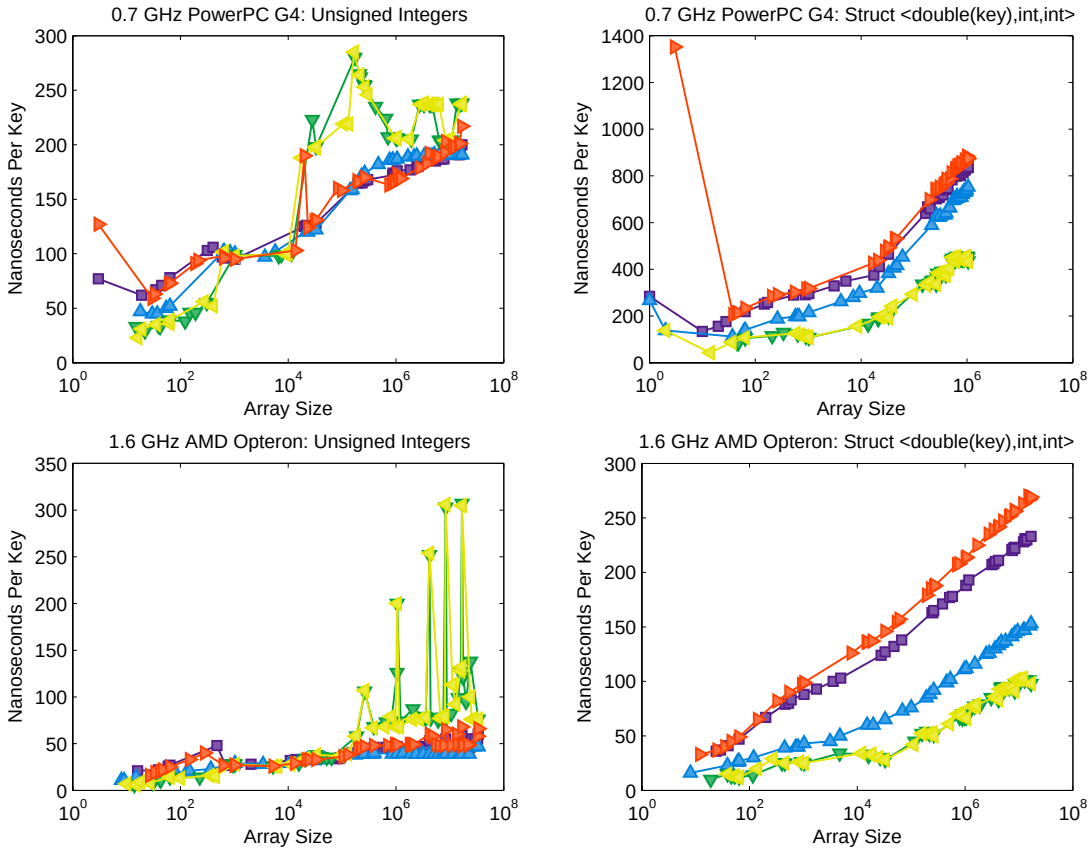

FIGURE 7.6. The performance of ATSL's generated algorithm on five distributions, continued from Figure 7.5

of quicksort for use on small input sizes. Under Microsoft's compiler, ATSL used mergesort for the smallest arrays and samplesort for slightly larger ones.

8. CONCLUSIONS

We have presented ATSL, a system for generating highly-tuned sorting subroutines. The overall structure of ATSL is similar to that of other automatic-tuning systems, such as ATLAS [24], PHiPAC [2], FFTW [8], and OSKI [23] (see also [4]). Like these systems, ATSL explores a large space of synthetic algorithms, generates some and measures their performance, and uses the results of the measurements to synthesize an efficient algorithm.

However, the performance of sorting algorithms is highly input-dependent, unlike the performance of dense-matrix algorithms and of Fourier transforms. Our approach has been to address that issue by using training inputs that represent several important input distributions. However, our results indicate that in some cases this approach is insufficient. The approach proposed by Li, Garzarán and

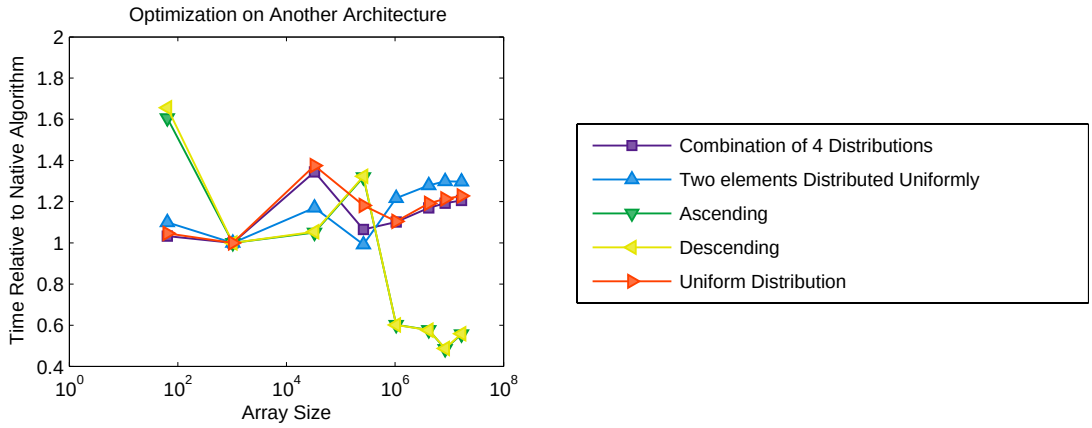

FIGURE 7.7. The performance of the algorithm that ATSL produces on a 0.7 GHz G4 machine, when used on a 0.5 GHz Pentium 3. The data type is unsigned integers. The graph shows the running time divided by the running time of the Pentium's native ATSL algorithm. Values above 1 mean that the G4 algorithm is slower than the Pentium's native algorithm, values lower than 1 mean that the G4 algorithm is faster.

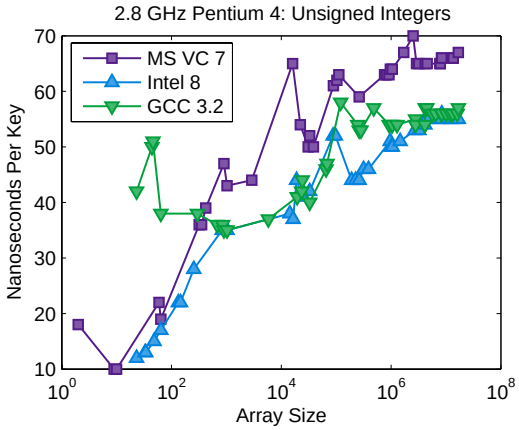

FIGURE 7.8. The performance of ATSL's generated algorithms under three different compilers on a single machine, a 2.8 GHz Pentium 4. The data type is unsigned integers and the data distribution is ATSL usual distribution.

Padua [15, 16] is more robust in this respect. They have also proposed an automatic tuning system for sorting algorithms, called XSORT, which computes an entropy metric for each input. The algorithm that their resulting sorting subroutine selects for a specific input depends on this metric.

Our approach and the approach of Li, Garzarán and Padua are largely complementary. They have focused mostly on the input-dependence problem in sorting algorithms and on using a genetic-search algorithm to synthesize an efficient algorithm. We have focused mostly on the a systematic characterization of the search space, described in Chapter 3. We have also spent a considerable amount of effort on ensuring that ATSL's resulting algorithms are significantly faster than hand-coded algorithms, and on the software-engineering aspects of ATSL (in particular, on simplifying the inclusion of new building blocks). We have also attempted to include in ATSL and in our benchmarking tests as many hand-coded algorithms in order to strengthen the claim that automatically-generated sorting algorithms can perform better than hand-coded and hand-tuned codes. Li, Garzarán and Padua have not included such codes in their evaluations.

Li et al. have not provided us with a copy of their system, so we cannot directly compare the performance of the algorithms that the two systems produce. However, from locating the platforms' STL algorithm in their performance graphs it seems that the performance of XSORT-produced algorithms and ATSL-produced algorithms is similar; both beat the STL by a significant factor. Also, both systems seem to select radixsort variants for large array sizes.

Most importantly, our results show that neither hand-tuning nor standard compiler optimization are sufficient to obtain a high-performance sorting algorithm. Sorting-specific automatic tuning and synthesis is also necessary. We have included in our system and in our experimental evaluation most of the high-performance hand-coded algorithms that have been produced in the last decade or so. Our results show that none of them is as fast as an automatically-tuned algorithm, often by a large margin. Our results are stronger in this respect than the results of Li, Garzarán and Padua, who compared their system's resulting algorithm to commercial and compiler-provided algorithms, but not to research codes produced by other groups.

Acknowledgement. Thanks to Maria Garzarán, Yishai Feldman, and Amiram Yehudai for helpful feedback on the manuscript. Thanks to all the authors who shared their hand-coded sorting codes with us. This research was supported in part by an IBM Faculty Partnership Award, by grants 572/00 and 848/04 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by grant 2002261 from the United-States-Israel Binational Science Foundation.

References

[1] Selim G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1990.

[2] Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel. The PHiPAC v1.0 matrix-multiply distribution. Technical Report UCB/CSD-98-1020, CS Division, University of California at Berkeley, October 1998.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[4] James Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, volume 93, February 2005.

[5] Gerth Stølting Brodal Rolf Fagerberg and Gabriel Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the 32nd International Colloquium on Automata, Languages, and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 576–588. Springer Verlag, 2005.

[6] Gerth Stølting Brodal Rolf Fagerberg and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments*, pages 4–17, New Orleans, January 2004.

[7] Edward H. Friend. Sorting on Electronic Computer Systems. *J. ACM*, 3(3):134–168, 1956.

[8] Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

[9] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–298, Washington, DC, USA, 1999. IEEE Computer Society.

[10] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[11] Dror Irony, Gil Shklarski, and Sivan Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. *Future Generation Computer Systems*, 20(3):425–440, April 2004.

[12] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[14] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31(1):66–104, 1999.

[15] Xiaoming Li, María Jesús Garzarán, and David A. Padua. A dynamically tuned sorting library. In *Proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, San Jose, California, June 2004. IEEE Computer Society.

[16] Xiaoming Li, María Jesús Garzarán, and David A. Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the 2005 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 99–110, San Jose, California, March 2005. IEEE Computer Society.

[17] Hosam M. Mahmoud. *Sorting: A Distribution Theory*. Wiley-Interscience, 2000.

[18] Dragan Mirkovic. Automatic performance tuning in the UHFFT library. In *Proceedings of the International Conference on Computational Science (ICCS), Part I*, volume 2073 of *Lecture Notes In Computer Science*, pages 71–80, San Francisco, 2001. Springer-Verlag.

[19] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In Susanne Albers and Tomasz Radzik, editors, *Proceedings of the 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796, Bergen, Norway, September 2004. Springer.

[20] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[21] Robert Sedgewick. *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison Wesley, 3rd edition, 1998.

[22] H. H. Seward. Masters thesis, m.i.t. digital computer laboratory report r-232. 1954.

[23] Richard Vuduc, James Demmel, and Katherine Yelick. An interface for a self-adapting sparse kernel library. Technical Report (*to* appear), University of California, Berkeley, Berkeley, CA, USA, September 2004.
[24] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
[25] Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, and Jeffrey Scott Vitter. Efficient sorting using registers and caches. *J. Exp. Algorithmics*, 7:9, 2002.
[26] J. W. J. Williams. ACM Algorithm 232: Heapsort. *Journal of the ACM*, 7(6):347–348, June 1964.
[27] Paul Youn. Serial and parallel execution of Funnel Sort. Final course project, available online at `http://theory.lcs.mit.edu/classes/6.895/fall03/projects/`, 2003.

Sivan Toledo, School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.
E-mail address: `stoledo@tau.ac.il`