Tuning Research Tools for Scalability and Performance: The NICAD Experience

James R. Cordy

School of Computing
Queen's University
Kingston, Ontario, Canada

Chanchal K. Roy

Department of Computer Science
University of Saskatchewan
Saskatoon, Sasksatchewan, Canada

Abstract

Clone detection is a research technique for analyzing software systems for similarities, with applications in software understanding, maintenance, evolution, license enforcement and many other issues. The NiCad near-miss clone detection method has been shown to yield highly accurate results in both precision and recall. However, its naive two-step method, involving a parsing first step to identify and normalize code fragments, followed by a text line-based second step using longest common subsequence (LCS) to compare fragments, has proven difficult to migrate to the efficiency and scalability required for large scale research applications. Rather than presenting the NiCad tool itself in detail, this paper focuses on our experience in migrating NiCad from an initial rapid prototype to a practical scalable research tool. The process has increased overall performance by a factor of up to 40 and clone detection speed by a factor of over 400, while reducing memory and processor requirements to fit on a standard laptop. We apply a sequence of four different kinds of performance optimizations and analyze the effect of each optimization in detail. We believe that the lessons of our experience in migrating NiCad from research prototype to production performance may be beneficial to others who are facing a similar problem.

Keywords: clone detection, longest common subsequence, optimization, NiCad

1. Introduction

The copy/paste/edit cycle is a common practice in software development, and as a result most software systems contain a significant number of similar code fragments, called *code clones* or simply *clones*. While the practice of code cloning can speed up and simplify software development, and in some cases can actually be beneficial [1, 2], it can also lead to software understanding and maintenance difficulties that can cause problems down the road [3].

Clone detection, or the analysis of source code for identical or similar fragments, has become a popular and practical application of source analysis to assist in finding and resolving such clones. A wide range of tools and techniques have been proposed for clone detection [4], and it is currently an active and popular area of source code analysis research. One of the most recent methods is the hybrid technique NiCad [5], which is the subject of this paper.

The NiCad prototype tool has been shown to yield high accuracy in both precision [5] and recall [6], and has proven itself practical in large initial research studies [7, 8]. However, at several days to process the Linux kernel, its performance leaves a lot to be desired. In this paper, rather than presenting the NiCad tool itself in detail, we address the performance issue head-on. We present a number of tuning steps, including memory-residence, file reduction, line-level hashing, and a custom optimization of the longest common subsequence (LCS) algorithm to reduce comparison time. The objective is to move from the initial internal rapid prototype to a scalable, efficient research tool for use by other researchers.

This paper is not about clone detection per se, and not about comparing or evaluating clone detectors. There are a number of other such studies already in the literature [9, 4] and no doubt there will be more in the future. This paper is about performance and scalability tuning, and in particular about tuning source code processing systems, as represented by clone detection tools. Our goal is to share our experience in tuning the performance and scalability of one particular source code analysis system, the NiCad clone detector, in the hopes that others may benefit from what we have learned about what matters, what worked for us, and what the relative effect of different tuning strategies may be expected to be. Tuning software systems in general and source analysis systems in particular is an art [10], and the more we know about what has worked in the past, the better we can practice that art in the future.

The remainder of this paper is structured as follows. In Section 2 we outline the problem we are attacking and the motivation for our tuning efforts. In Section 3 we review the basics of the NiCad clone detection method and the structure

and properties of the original NiCad clone detector, including its prototype implementation and poor performance. In Section 4 we outline the design of our tuning experiment and introduce the set of systems we use as a joint benchmark for measuring our performance improvements and original NiCad's performance on them. Section 5 introduces our first tuning step, moving all comparisons from files to memory, and measures the improvements we observed. Section 6 measures the effect of our second step, amalgamating all fragment files into one to avoid system overhead. Section 7 introduces and measures the effect of our third step, using a hash table to convert text lines to hash codes, and finally Section 8 describes a specialization of the LCS length algorithm to speed fragment comparison and measures its effect on overall performance. Section 9 reviews the overall effect of our tuning efforts on performance and scalability, and analyzes the relative effect of our various optimizations. Section 10 briefly admits that there certainly are other clone detection methods with even better performance. Section 11 outlines the lessons we have learned from this experience and the possible implications for tuning other tools. Our conclusions (Section 12) wrap up the paper with further plans for the future of NiCad.

2. Motivation

Clone detectors face three fundamental challenges: *accuracy*, the extent to which a clone detector finds cloned code with high precision (i.e., returns few false positives) and recall (i.e., returns few false negatives), *performance*, the speed with which the detector can return its results, and *scalability*, the extent to which the detector can handle increasingly large systems. Meeting all three of these is difficult, and many creative ideas, such as suffix trees [11], metrics [12], parallel computation [13] and dynamic clustering [5] have been proposed to improve speed and other attributes.

In our previous research on the NiCad clone detection tool [5], we have chosen to concentrate primarily on accuracy at the cost of scalability and performance. Recognizing that the accuracy of clone detectors can be subjective, we developed a method-independent copy/paste/edit theory of clone creation by programmers that provides a formal definition for clones that can be used to objectively assess accuracy [14]. Based on this theory, we validated both the high precision of the NiCad method using a combination of back-checking using raw *diff* and a custom interface that allows for practical total hand validation [5], and the high recall of the NiCad method using thousands of artificially generated clones in a mutation-based automated assessment framework [6].

Now that we are confident about accuracy, our attention turns to issues of scalability and performance. While NiCad is designed to scale using parallelization, and has been used for one or two initial large scale research studies [7], its poor performance (e.g., a few days to process the Linux kernel, even running 4-way parallel) makes it impractical for wider research use, and analysis of systems ten times larger than Linux is simply infeasible.

Our specific motivation comes from a range of new requirements for clone detectors, specifically: (i) the desire to bring clone detection research tools "to the masses" on their own laptop PC, using reasonable resources on a modest single processor; (ii) the desire to embed clone detection as an immediate service in code development environments, with rapid incremental clone analysis as users create or edit code; (iii) the desire to handle truly large code bases, such as the entire eight DVD Debian source distribution, in order to audit other systems for copyright or GPL violations; or the entire 48 releases of the FreeBSD kernel, in order to track long-term code evolution. All three of these desires require high speed and scalability, and the first two additionally require a modest memory footprint. The optimizations outlined here have achieved all three of these goals [15, 16, 17].

3. NiCad

The NICAD (a dyslexic acronym for *Accurate Detection of Near-miss Intentional Clones*) clone detector is a hybrid parser-based / text comparison clone detection system. It is designed to combine the high precision of language-sensitive parser-based clone detection methods with the high recall of language-independent text- or token-based systems, in order to increase overall accuracy. At present it can process programs in four different languages: C, Java, C# and Python.

The original NiCad design grew out of a multi-language (HTML, ASP, JSP) web application refactoring project, which required a clone detection method that was at once language structure sensitive and applicable across a range of languages, while allowing for unexpected nonstructural editing changes in the code [18]. Input size was not originally an issue, since we were processing only static web pages of relatively modest size.

The simple two-phase design we adopted (Figure 1) focuses on simplicity, accuracy and flexibility in near-miss clone detection. In the first phase, a language-sensitive island parser implemented in TXL [19] is used to recognize and extract all fragments of the desired granularity (e.g., functions, blocks or statements) for comparison as clones. Extraction uses the TXL non-terminal extraction built-in function, which creates a list of all subtrees of a given non-terminal type (e.g.

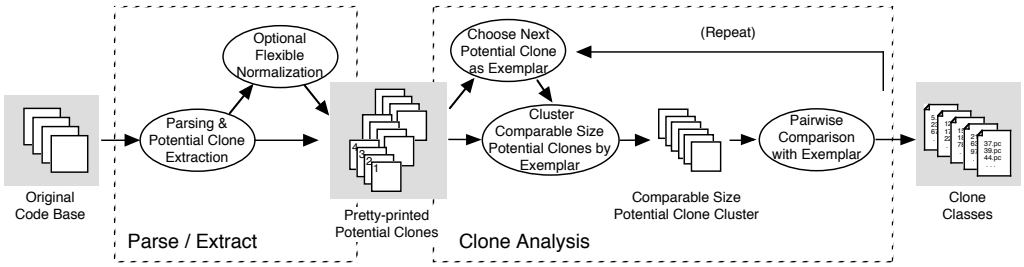

Figure 1: NiCad process architecture.

function, block, statement) in the parse tree of a given input. The structure of the code recognized by the parser is preserved in the text of the extracted fragments, called *potential clones*, using formal indentation and line break rules ("pretty-printing") to encode language structure. The result is the set of all code fragments of the given granularity from all the source code files of a system, each pretty-printed, comment-stripped and stored in a separate file. Figure 2 gives an example of a small C source file and the potential clones that would be extracted from it at the function granularity.

In the second phase, a language-independent near-miss comparator compares the lines of text in pairs of potential clones using the *diff* algorithm, recognizing as clones those extracted fragments whose pretty-printed text form differs by less than a given threshold (e.g., 30%, representing a maximum difference of 3 lines in every ten). If the threshold is 0%, allowing for no differences at all, matching fragments are called *exact clones*, referred to in the clone research community as Type 1 clones. If the threshold is greater than 0%, then the matching fragments are called *near-miss* clones, in that they are exact copies except for a few lines of change. These are referred to in the cloning community as Type 3 clones. Figure 3 shows an example of two potential clone files that are near-miss clones at the 30% difference threshold but not at 20%.

The NiCad method has been generalized to allow for an optional middle phase, normalization. In normalization, potential clone files can be pre-processed for renaming, filtering or abstraction before comparison. Clones that are found by exact matching of normalized fragments are referred to in the cloning community as parameterized or Type 2 clones, which allow for changes in identifiers, literal values or other items within matching source lines, but do not allow for changes between lines. Since our optimizations have the same effect with or without normalization, we ignore it in this paper.

In the original rapid prototype implementation of NiCad, a TXL [19] program is used to extract and pretty-print the potential clones to a set of separate potential

```
/* Example C Source File */
void asn1_table_unset
       (hash_t *table,char *key){
    ssize_t klen=strlen(key);
    asn1_t *asn1=
      hash_get(table, key, klen);
    if(!asn1){
        return; /* Give up */
    }

    /* Free hash pointer */
    free(asn1);
    /* Clear hash entry */
    hash_set(table,key,klen,NULL);
    }

static const char *key_types[] =
/* Only two */ {"RSA", "DSA"};

const char *asn1_keystr(int keytype){
    /* Return if past max */
    if (keytype >= SSL_AIDX_MAX) {
        return NULL; }

    return asn1_key_types[keytype]; }

const char *table_keyfmt(pool_t *p,
         char *id, int keytype) {

  /* Make standard key format */
  const char *keystr =
      asn1_keystr(keytype);
  return pstrcat(p, id, ":",
      keystr, NULL);
}
```

```
void asn1_table_unset (apr_hash_t *table, char *key)
{
    ssize_t klen = strlen (key);
    asn1_t *asn1 = hash_get (table, key, klen);
    if (!asn1) {
        return;
    }
    free (asn1);
    hash_set (table, key, klen, NULL);
}
```

```
const char *asn1_keystr (int keytype)
{
    if (keytype >= SSL_AIDX_MAX) {
        return NULL;
    }
    return asn1_key_types [keytype];
}
```

```
const char *table_keyfmt (pool_t *p, char *id, int keytype)
{
    const char *keystr = asn1_keystr (keytype);
    return pstrcat (p, id, ":", keystr, NULL);
}
```

Figure 2: An example C source file and extracted potential clones at the function granularity.

```
void leftone (apr_hash_t *table, char *key)
{
    ssize_t klen = strlen (key);
    asn1_t *asn1 = hash_get (table, key, klen);
    if (!asn1) {
        return;
    }
    free (asn1);
    hash_set (table, key, klen, NULL);
}
```

```
void rightone (hash_t *table, char *key)
{
    ssize_t klen = strlen (key);
    asn1_t *asn1 = hash_get (table, key, klen);
    if (!asn1) {
        return (NULL);
    }
    asn1 = NULL;
    hash_set (table, key, klen, NULL);
}
```

Figure 3: A pair of C function potential clones which are near-miss clones at the 30% difference threshold level but not at 20%. The three differing lines are highlighted in boldface.

clone files, and an optimized open-source Perl implementation of the *diff* longest common subsequence (LCS) algorithm is used to compare potential clone files as pairs. A Perl driver script dynamically chooses comparable sets of potential clone files using an exemplar-based strategy that makes one pass over a sorted list of the sizes of the potential clone files.

Potential clone files are first sorted from largest to smallest by number of lines. The largest potential clone is then chosen as an exemplar. The comparable files are then those whose difference in number of lines from the examplar is within the difference threshold. For example, if the size of the exemplar is 10 lines and the near-miss difference threshold is 30%, then all potential clones between 7 and 13 lines form the comparable set. Once the exemplar is compared to these and clones of it identified, they are removed from the comparison set and the next largest remaining potential clone is chosen as exemplar, and so on. The script takes

advantage of the number of processor cores on the host machine to parallelize comparison of the sets as much as possible.

While this prototype implementation has proven practical enough to support our own research studies of cloning in open source software [7, 8], its performance has been a problem, and is a serious barrier to its use as a general research tool by others. For example, using NiCad to detect near-miss function clones in a relatively modest mid-sized system such as Apache *httpd* at a 30% difference threshold takes about 15 minutes on a standard desktop PC. While one may apologize that, while slow compared to almost any other method, this is still usable given the high accuracy of NiCad, it does not scale up – when *httpd* is processed at the finer granularity of C blocks for the same difference threshold, the time required grows to over an hour. Worse, for large systems, performance is a problem even at the function granularity. For example, the total time for NiCad to extract and find near-miss function clones in version 2.6.24.2 of the Linux kernel at a 30% difference threshold on a quad-core 8 Gb compute server is almost a week. Clearly such poor performance cannot serve us for finer granularities, longitudinal studies, entire distribution code bases, real time personal clone detection or rapid turnaround incremental analysis.

4. The Experiment

As a result of the observations above, we undertook to investigate methods for tuning the performance of the NiCad tool to reasonable levels. The basic questions to begin with were: Where does the time go? Are there any obvious opportunities ("low-hanging fruit") for improving performance? Can we learn anything that others might use to tune their systems?

While NiCad is built in two stages, the first, implemented in TXL using the TXL parser, is for the most part already tuned as a result of more than two decades of TXL use on large scale problems. Initial measurements indicated that the vast majority of the original NiCad time (more than 85% for *httpd* up to more than 99% for Linux) was spent in fragment comparison. Thus we have concentrated most of our tuning efforts on clone analysis and comparison itself – problems that are faced to some extent by all clone detection methods.

Based on an initial instrumentation of the Perl-based NiCad process, we decided on six basic "research" questions for our tuning experiment:

1. Would changing to a compiled language from an interpretive one significantly change performance?

2. What would be the effect of using memory-resident rather than file-based comparison of potential clones?
3. What is the effect of file system performance and file representation on overall performance?
4. Can line-level hashing be used to improve comparison time and space?
5. Can the LCS length algorithm be optimized for near-miss clone detection?
6. And finally, what is the relative effect of each of these techniques on overall performance?

In order to answer these questions, we systematically applied each of these potential optimizations to the NiCad implementation, one by one. In each case we tracked the effect on performance. Since true "memory resident" data (e.g., direct memory-mapped arrays) is not a guaranteed property of interpretive languages such as Perl, and since the challenge of recoding the complex Perl scripts running old NiCad was formidable, we bundled the first two optimizations into one.

In order to track performance, we chose a suite of example applications chosen from the four languages that NiCad can handle so far, and formed a baseline of the times that our original NiCad prototype takes to process them at two different granularities (Table 1). The example systems are mid-sized representatives of each of the languages, and in the case of C and Java systems (*httpd, postgresql, eclipse-jdtcore, jEdit*, etc.) have been chosen because of their familiarity as examples in the clone detection community. (The Linux kernel, which due to its large size would dominate the results, is considered separately – see Section 9).

Since the clone detection results from all these systems have already been reported and their accuracy validated in our previous experiments [7, 8], we will not repeat that data here. We simply say that since the method and algorithm remains the same, none of our optimizations changes any of our clone detection results, other than a tiny rounding effect due to differences in floating point representation between Perl and compiled code. At each stage we used the results of the original NiCad implementation on these systems as a regression test to insure that our optimizations did not introduce any unexpected behaviour.

In order to reduce the number of free variables in the experiment, we have chosen to show results only for the case of near-miss exact (Type 3) clones – that is, those without renaming, filtering, abstraction or other normalization. Since normalization affects only extraction and is not affected by any of our optimizations, we are confident that our results are also valid for the normalized case.

Using Table 1 as a baseline, in each stage of the tuning experiment we used the sum of the total processing times for all of the systems together ("Total all

(a) Functions

					Elapsed time (sec)				
Language	**System**	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
C	httpd	539	275,255	5,752	148.41	148.62	313.49	539.79	695.97
	postgresql	322	201,686	4,685	130.02	132.10	249.92	434.17	574.96
C#	Castle	2,419	130,565	9,529	223.88	271.13	379.41	647.83	801.62
	RssBandit	316	166,495	4,580	195.41	54.44	86.81	157.53	212.91
Java	eclipse-jdtcore	741	147,634	7,696	168.35	150.41	266.79	442.81	551.73
	jEdit	539	173,792	6,251	151.08	106.21	185.07	328.16	422.72
	JHotDraw	285	40,063	2,536	39.78	11.13	14.61	27.65	38.83
Python	Django	1,343	140,117	7,084	168.96	179.93	247.16	457.79	604.07
	Eric	743	196,513	7,659	257.57	184.88	306.94	525.17	688.63
Total All Systems		7,247	1,472,120	55,772	1,483.46	1,238.85	2,050.20	3,560.88	4,591.43

(b) Blocks

					Elapsed time (sec)				
Language	**System**	**#Files**	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
C	httpd	539	275,255	24,335	633.36	1,175.66	1,741.77	2,731.13	3,617.39
	postgresql	322	201,686	13,972	508.27	995.51	1,404.64	1,983.93	2,600.88
C#	Castle	2,419	130,565	23,311	449.03	640.09	881.07	1,314.93	1,658.40
	RssBandit	316	166,495	16,145	684.87	357.47	548.29	854.47	1,018.05
Java	eclipse-jdtcore	741	147,634	23,438	509.59	874.97	1,385.44	2,010.98	2,549.53
	jEdit	539	173,792	15,679	361.12	530.23	813.72	1,244.30	1,525.93
	JHotDraw	285	40,063	4,543	67.37	23.40	33.12	58.48	75.92
Python	Django	1,343	140,117	20,583	430.77	868.64	1,257.53	1,973.27	2,622.24
	Eric	743	196,513	26,870	837.46	1,433.57	2,114.20	3,407.49	4,475.06
Total All Systems		7,247	1,472,120	168,876	4,481.84	6,899.54	10,179.78	15,578.98	20,143.38

Table 1: Original NiCad prototype extraction and analysis times for the benchmark systems.

systems") at each of four near-miss thresholds, 0% (exact clone), 10% (1 line in 10 difference allowed), 20% and 30%. We chose not to track CPU time, but rather elapsed real (clock) time, since clone detector users will be interested in real world throughput, not just CPU use. Throughout this paper all references to "time" refer to elapsed clock time as measured by the Linux *time* command on a quiet machine running only the NiCad process. While these measurements are not exact, they vary by less than 10% in repeated runs, and we have reported times for the average of five runs. While the test machine itself is a 2.66 GHz quad-core Intel processor with 8 Gb of memory, all measurements have been made using a single core in a 2 Gb memory partition. Except in the case of the first optimization, reported improvements are incremental, that is, they are speedup factors relative to the previous optimization rather than the original baseline.

While we tried to track improvements in memory use as well, they are difficult to measure and reproduce in the Linux operating system environment, so we only present them in qualitative summary terms. All versions handle all of these example systems in less than 2 Gb, and our final version uses less than 2 Gb even

for Linux and other systems of over a million lines, making it easily practical on laptop computers from a memory standpoint.

5. Reimplementation Using Memory-resident Comparison and Compiled Code

As outlined in Section 3 above, the original NiCad prototype uses large numbers of small files to represent and compare fragments (potential clones). Because modern operating systems such as Linux optimize repeated access to the same file using memory buffering, for small systems this does not significantly affect performance since on second and subsequent accesses the potential clone files are already buffered in memory. However, there is nevertheless a significant amount of time spent in system overhead accessing these small files, and for mid- to large-sized systems this overhead rapidly becomes the dominant factor, accounting for a large part of the observed elapsed times in Table 1.

Thus our first discovery in looking for "low hanging fruit" to optimize is the observation that the total text of all potential clone files, even for all of the 156,000 functions of Linux, is only about 127 Mb – making it practical to avoid most of the file access overhead by reading all of the potential clone files into memory once, and then comparing them directly in memory as needed.

Our first optimization of NiCad was therefore a reimplementation using native data arrays in a compiled language, Turing+ [20], a type-safe systems programming variant of Pascal. We chose to use Turing+ because of its generated code efficiency (it uses the gcc code generator), its support for native memory arrays (like C, but subscript-safe, making it easier to debug), and our local familiarity with it because it is the implementation language used for TXL.

Table 2 shows the results of reimplementation using memory-resident fragment comparison in Turing+. Conversion to reading the potential clone files once and memory-resident comparison in a compiled language yielded a clone analysis speed increase of a factor of a minimum of 57 times for all difference thresholds, and over 500 times for the case of exact clones.

The effort to undertake this change was significant, requiring about one programmer-month to design and code the new implementation using the original NiCad as a reference. Much of the time was spent in understanding how the original Perl code did what it did, a job that was made more difficult because of the code obscurity introduced by its previous parallelization optimization.

				Elapsed time (sec)				
All Systems	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Original	7,247	1,472,120	55,772	1,483.46	1,238.85	2,050.20	3,560.88	4,591.43
Memory Res.				1,483.46	2.35	27.52	54.49	80.56
Speed up factor				1	527	75	65	57
	#Files	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Original	7,247	1,472,120	168,876	4,481.84	6,899.54	10,179.78	15,578.98	20,143.38
Memory Res.				4,481.84	13.10	118.87	230.97	329.94
Speed up factor				1	527	86	67	61

Table 2: Extraction and clone analysis times in seconds after reimplementation to use memory-resident comparison and compiled code.

6. File Reduction

Our next optimization continues on the theme of file access. While we have reduced file access by reading potential clone files into memory only once, we still have to pass all of these files from the extraction phase to the clone analysis phase. For small systems like *JHotDraw* which has only a few thousand potential clones (functions or blocks), writing and reading them does not present a significant overhead. However, for large systems, there can be hundreds of thousands or even millions of these potential clone files. For example, there are over 156,000 nontrivial functions in the Linux kernel, and over 2 million in the 48 releases of the FreeBSD kernel. The file systems of modern operating systems such as Linux's *ext3* and Windows' *NTFS* are simply not optimized to handle such large numbers of small files well, particularly if they are stored in a single directory. Thus the system overhead and elapsed times of NiCad can grow significantly with the size of the system, due not to its own processor use, but to the file system overhead incurred by writing and reading so many small files.

One solution to this problem which has been suggested by others is to change the file system. For example, we could change operating systems (Mac OS X has less of this problem), or we could reconfigure Linux to use, say, XFS or Apple's HFS+ file system, both of which handle directories with large numbers of files more efficiently. Yet another solution, used by Linux archival systems, is to organize the large sets of small files into trees of subdirectories to avoid the inefficient large directory problem.

In our case we have control of both the source (the parser/extractor) and the sink (the clone analyzer) of the large numbers of small files, and our only issue is communicating the outputs of one to the inputs of the other. We can not simply pass the files in memory, both because repeated extraction is expensive, and because we may want to run many different code analyses on a single extraction.

```
<source file="example/egcfile.c" startline="2" endline="16">
file=void asn1_table_unset (apr_hash_t *table, char *key)
{
    ssize_t klen = strlen (key);
    asn1_t *asn1 = hash_get (table, key, klen);
    if (!asn1) {
        return;
    }
    free (asn1);
    hash_set (table, key, klen, NULL);
}
</source>
<source file="example/egcfile.c" startline="21" endline="26">
const char *asn1_keystr (int keytype)
{
    if (keytype >= SSL_AIDX_MAX) {
        return NULL;
    }
    return asn1_key_types [keytype];
}
</source>
<source file="example/egcfile.c" startline="28" endline="36">
const char *table_keyfmt (pool_t *p, char *id, int keytype)
{
    const char *keystr = asn1_keystr (keytype);
    return pstrcat (p, id, ":", keystr, NULL);
}
</source>
```

Figure 4: Example single XML file representation of the extracted potential clones of Figure 2.

For example, we may want to experiment with varying normalizations and near-miss thresholds for clone detection, or use other tools such as feature and concept analyzers. Thus in our next optimization, we changed the extraction process to output all extracted potential clones from all source files to a single sequential file in XML format. Figure 4 shows an example XML potential clones file corresponding to the three potential clones files shown in Figure 2. Each parsed and pretty-printed potential clone is enclosed in `<source>` tags giving its original file and line number information, all in a single file that is sequentially read into the clone detection process. Since all modern operating systems are highly optimized for sequential read and write of large files, this allows us to use the file system in its most efficient mode, first sequentially writing all the potential clones from the extractor and then sequentially reading them into the clone detector.

This change should affect both the extraction time (since the parser/extractor writes only one file, not large directories of small ones) and the comparison time (since clone analysis now reads only one large sequential file rather than thousands of small ones). Table 3 shows the results of our change to use a single file for all potential clones. Although this change has only a tiny effect on clone analysis time, it yielded a speed increase in extraction time of a factor of 3.90 and 7.92 for functions and blocks respectively.

The difference in effect of this optimization on the two phases is not simply due to the difference in writing vs. reading of the many files, but rather to the

				Elapsed time (sec)				
All Systems	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Memory Res.	7,247	1,472,120	55,772	1,483.46	2.35	27.52	54.49	80.56
Single File				380.00	1.39	26.75	54.37	80.14
Speed up factor				3.90	1.69	1.03	1.00	1.01
	#Files	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Memory Res.	7,247	1,472,120	168,876	4,481.84	13.10	118.87	230.97	329.94
Single File				566.20	9.86	116.2	227.64	327.03
Speed up factor				7.92	1.33	1.02	1.01	1.01

Table 3: Extraction and clone analysis times in seconds after reduction to single file representation of potential clones.

Linux operating system optimization that keeps recently accessed files buffered in memory, thus charging all of the access overhead to the producer of the files (extraction) rather than the consumer (clone analysis). If in the original system clone analysis had been run again later on the same extracted files, when they had not been recently accessed, then a similarly large speed increase would have been seen for clone analysis.

The effort to implement this optimization was not large. It took less than two programmer-days to modify the driver program for the TXL parse-extract to concatenate all its outputs onto one file and the Turing+ clone analysis program to read all the potential clones from one file rather than one per file. A regression test of all of our previous results for the test systems insured that we had introduced no unexpected changes in behaviour.

7. Text Line Hashing

Since differences in commenting, formatting, line boundaries and spacing are eliminated by the NiCad potential clone extraction process and the granularity of comparison is chosen by the extractor as part of its pretty-printing, potential clone comparisons are carried out at the line-of-text level, where each line to compare is a normalized, pretty-printed line of text to be compared only for equality. Each two potential clones of comparable size are compared line-by-line using a longest common subsequence (LCS) length algorithm, comparing pairs of lines textwise. Naturally, these comparisons dominate the cost of the LCS algorithm.

In compilers and interpreters, text comparison of identifiers is optimized and avoided using hash tables, which reduce the text comparison for equality of identifiers to integer comparison. Since we are also only interested in equality (of normalized, pretty-printed text lines), an obvious optimization for us is to do similarly, using a hash table to convert lines to integer hash codes for comparison.

				Elapsed time (sec)				
All Systems	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Single File	7,247	1,472,120	55,772	380.00	1.39	26.75	54.37	80.14
Line Hash				380.00	0.96	10.79	20.75	30.40
Speed up factor				1.00	1.45	2.48	2.62	2.64
	#Files	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Single File	7,247	1,472,120	168,876	566.20	9.86	116.2	227.64	327.03
Line Hash				566.20	2.91	42.41	81.79	115.65
Speed up factor				1.00	3.39	2.74	2.78	2.83

Table 4: Extraction and clone analysis times in seconds after addition of text line hashing.

Fortunately, we already had a module that implements generalized text hashing in the TXL source transformer, which also happens to be implemented in Turing+. Having been tuned over decades of TXL use processing source code in a range of languages, it is also a very fast and efficient hashing algorithm. Our next optimization therefore was to adapt this module to implement a line-of-text hash table for NiCad, reducing all comparisons in the LCS algorithm to integer equality. In retrospect, this is an obvious change, since in NiCad lines effectively play the role of tokens in token-based methods, all of which use hash codes.

The result of this optimization is shown in Table 4. While there is a slight initial cost to entering potential clone text lines into the hash table on reading them in, clearly it is more than made up for in the increased speed of line comparison. As we can see, this optimization has a huge effect in the simple case of comparison for exact clones.

Because we were adapting an existing text hashing module from the TXL implementation, the effort to introduce line hashing required no more than two or three programmer-days to adapt the module and modify the input of potential clone text lines to use it. Regression testing of previous results once again made sure that functional behaviour had not been affected.

8. Bounded LCS

Our final optimization is algorithmic. NiCad's near-miss criterion is based on a longest common subsequence (LCS) length difference threshold, called the UPIT (unique percentage of items threshold). At a UPIT of 0.10, for example, up to 10% of the normalized pretty-printed lines between two fragments (potential clones) may differ for them to be considered near-miss clones. This criterion is implemented by the formula [5]:

$$\frac{uniqueLines(PC1)}{totalLines(PC1)} \leq UPIT, \quad and \quad \frac{uniqueLines(PC2)}{totalLines(PC2)} \leq UPIT$$

where *uniqueLines(PC1)* is defined as :

$$totalLines(PC1) - LCS\,length(PC1, PC2)$$

and similarly for PC2, and *LCSlength(PC1,PC2)* is the length of the longest common subsequence of the lines in PC1 and PC2.

In the original NiCad prototype, the threshold is implemented by computing the LCS length using an open source Perl implementation of the Unix *diff* algortihm [21] and then comparing the difference between the number of lines in the LCS with the numbers of lines in each of the potential clones. In our Turing+ implementation, the threshold is computed using a direct translation of the standard dynamic programming solution for LCS length [22] (Figure 5).

Since the cost of near-miss clone detection is dominated by the computation of the LCS length (or similar measure), the question arises, can we specialize the LCS length algorithm to improve our overall performance? As in many applications of LCS length, we are really only interested in the question of whether two potential clones are close enough according to the threshold. In general, the vast majority of the comparisons we make will fail – that is, the two fragments will not be clones. Thus performance is dominated by LCS length comparisons that fail. The earlier we recognize that we are going to fail, the faster we can make the near-miss decision.

Figure 5 also shows the simple changes for a new algorithm for LCS length, "bounded LCS length" that takes this into account. We have added an upper limit to the LCS difference, which limits the number of items (in our case lines) that may be different before we give up because it is hopeless. The bound is precomputed from the lengths of the fragments to be compared. For example, if the fragments to be compared are 9 and 10 lines respectively, and the threshold is 0.20 (20%), then the bound is 2 lines – that is, if the fragments differ by more than 2 lines then they are surely not near-miss clones. This basic idea was first described by Hirschberg [23] in 1977, and we have simply adapted it to our algorithm.

In general, we can compute the bound using the formula:

$$round(max(totalLines(PC1), totalLines(PC2)) \times UPIT)$$

The bounded LCS algorithm takes this information into account by checking after processing each row (corresponding to a line of the first fragment) whether we already know that we have exceeded the bound. Because the dynamic programming algorithm accumulates the LCS length in the last element of each row of the comparison matrix, we can check this simply by subtracting the row number (number of lines checked so far) from the last element of the row (the length of the LCS so

```
% Standard dynamic programming version of the LCS length algorithm
var dpmatrix : array 0 .. maxclonelines, 0 .. maxclonelines of int

function lcs (pc1, pc2 : PC, m, n: int, difflimit: int ) : int
   for i : 0 .. m
      dpmatrix (i, 0) := 0
   end for
   for j : 0 .. n
      dpmatrix (0, j) := 0
   end for

   for i : 1 .. m
      for j : 1 .. n
         if lines (pc1.firstline + i) = lines (pc2.firstline + j) then
            dpmatrix (i, j) := dpmatrix (i - 1, j - 1) + 1
         elsif dpmatrix (i - 1, j) >= dpmatrix (i, j - 1) then
            dpmatrix (i, j) := dpmatrix(i - 1, j)
         else
            dpmatrix (i, j) := dpmatrix (i, j - 1)
         end if
      end for
      % Optimize by cutting off when it's hopeless
      if i - dpmatrix (i, n) > difflimit then
         result 0
      end if
   end for

   result dpmatrix (m, n)
end lcs
```

Figure 5: LCS length in Turing+, and the changes for the bounding optimization.

far), and comparing this to the bound (Figure 5). If the bound has been exceeded, we immediately return zero from the function, corresponding to the case where no lines match at all. This will certainly cause the near-miss criterion to fail at any difference threshold.

On the face of it, the bounded LCS function now completely implements the near-miss test. However, the bound we computed above is actually not an exact representation of the near-miss formula. Rather, it is rounded up to the closest exact number for the length of both fragments, and so we must still check the near-miss formula on return to yield results identical to the unbounded LCS.

Table 5 shows the effect of using the bounded LCS algorithm in place of the original LCS length in our near-miss comparisons. Of course there is no effect on the exact clone (0%) case because it does not use LCS.

Clearly the effort to introduce this optimization was minimal – only about one programmer-day was used to make the minor changes to add the threshold parameter to the LCS algorithm and modify the call to it to compute and pass the threshold for each pair of compared potential clones. As always, regression testing made sure we had not changed behaviour.

				Elapsed time (sec)				
All Systems	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Line Hash	7,247	1,472,120	55,772	380.00	0.96	10.79	20.75	30.40
Bounded LCS				380.00	0.96	2.55	6.10	11.34
Speed up factor				1.00	1.00	4.23	3.40	2.68
All Systems	**#Files**	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
Line Hash	7,247	1,472,120	168,876	566.20	2.91	42.41	81.79	115.65
Bounded LCS				566.20	2.91	11.25	28.21	49.57
Speed up factor				1.00	1.00	3.77	2.90	2.33

Table 5: Extraction and clone analysis times in seconds after bounded LCS optimization.

9. Summary and Analysis

Table 6 summarizes our detailed results for all systems in our test set. Since we have not changed the parser / extractors for any language, the improvement in extraction speed is consistent across all systems, and since the near-miss clone analysis is language independent, we see similar improvements as a result of our optimizations for all systems and languages. Total clone detection times have improved radically – for example, *httpd* has gone from a total of 148.41 + 695.97 seconds = about 14 minutes (Table 1) to 33.95 + 3.06 = 37 seconds at the 30% difference threshold level. Clone analysis alone has improved even more, going from 695.97 seconds (11 and a half minutes) to 3.06 seconds for a near-miss threshold of 30%. The improvement in overall time for all of the systems together is shown at the bottom of the table.

The preceding sections have addressed each of our original research questions, with the exception of the last one: "What is the relative effect of each of these techniques on overall performance?" In order to consider this question, we will view our improvements using two overall performance metrics: (i) overall processing speed in lines per second for the entire process, including both parsing/extraction and clone analysis, and (ii) clone detection speed in fragments per second for the clone analysis part alone at various near-miss thresholds (Table 7). We consider each of these metrics at the two most common clone granularities – functions and blocks. In reading these numbers, recall that all runs measured elapsed clock time using a single core of a 2.66 GHz Intel quad-core processor in a 2 Gb partition.

Overall Processing Speed. Figure 6 shows the improvements in overall lines per second processing speed for parse/extract alone and parse/extract plus clone analysis for our four test thresholds at each granularity, attributed by proportion contributed by each optimization. As we can see, the change to single file representation dominates the improvements in overall processing speed. For near-miss clone analysis, the change to memory residence of potential clones has the second

(a) Functions

					Elapsed time (sec)				
Language	**System**	**#Files**	**#Lines**	**#Fcns**	**Extraction**	**0%**	**10%**	**20%**	**30%**
C	httpd	539	275,255	5,752	33.95	0.14	0.49	1.46	3.06
	postgresql	322	201,686	4,685	27.22	0.10	0.39	1.00	2.02
C#	Castle	2,419	130,565	9,529	84.64	0.16	0.39	0.84	1.33
	RssBandit	316	166,495	4,580	23.42	0.09	0.15	0.27	0.46
Java	eclipse-jdtcore	741	147,634	7,696	23.09	0.12	0.35	0.85	1.54
	jEdit	539	173,792	6,251	22.43	0.09	0.26	0.50	0.98
	JHotDraw	285	40,063	2,536	6.44	0.03	0.04	0.07	0.10
Python	Django	1,343	140,117	7,084	57.18	0.10	0.22	0.48	0.69
	Eric	743	196,513	7,659	101.62	0.13	0.29	0.65	1.17
Total All Systems		7,247	1,472,120	55,772	380.00	0.96	2.55	6.10	11.34
Original Total					1,483.46	1,238.85	2,050.20	3,560.88	4,591.43
Speedup Factor					3.90	1,291.81	804.00	583.37	404.92

(b) Blocks

					Elapsed time (sec)				
Language	**System**	**#Files**	**#Lines**	**#Blocks**	**Extraction**	**0%**	**10%**	**20%**	**30%**
C	httpd	539	275,255	24,335	35.85	0.45	2.43	6.72	12.37
	postgresql	322	201,686	13,972	34.86	0.31	1.81	4.48	7.84
C#	Castle	2,419	130,565	23,311	84.27	0.39	1.13	2.38	3.99
	RssBandit	316	166,495	16,145	19.52	0.28	0.65	1.46	2.30
Java	eclipse-jdtcore	741	147,634	23,438	23.26	0.34	1.56	3.66	6.08
	jEdit	539	173,792	15,679	20.99	0.25	0.87	2.12	3.63
	JHotDraw	285	40,063	4,543	6.20	0.05	0.09	0.16	0.22
Python	Django	1,343	140,117	20,583	90.59	0.32	0.70	1.81	3.51
	Eric	743	196,513	26,870	250.67	0.53	2.02	5.45	9.64
Total All Systems		7,247	1,472,120	168,876	566.20	2.91	11.25	28.21	49.57
Original Total					4,481.84	6,899.54	10,179.78	15,578.98	20,143.38
Speedup Factor					7.92	2,367.72	905.27	552.29	406.34

Table 6: Final tuned NiCad extraction and clone analysis times in seconds for benchmark systems.

Functions	**Extraction**	**0%**	**10%**	**20%**	**30%**
Original (sec)	1,483.48	2,722.31	3,533.66	5,044.34	6,074.89
Lines/sec	992.34	540.76	416.60	291.84	242.33
Frags/sec		45.02	27.20	15.66	12.15
Final (sec)	380.00	380.96	382.55	386.10	391.34
Lines/sec	3,874.01	3,864.25	3,848.18	3,812.76	3,761.75
Frags/sec		58,156.41	21,871.37	9,136.96	4,918.60

Blocks	**Extraction**	**0%**	**10%**	**20%**	**30%**
Original (sec)	4,481.84	11,381.38	14,661.62	20,060.82	24,625.22
Lines/sec	328.46	129.34	100.41	73.38	59.78
Frags/sec		24.48	16.59	10.84	8.38
Final (sec)	566.20	569.11	577.45	594.41	615.77
Lines/sec	2,600.00	2,586.69	2,549.37	2,476.62	2,390.69
Frags/sec		57,953.33	15,017.87	5,986.81	3,406.61

Table 7: Original and final extraction and overall processing (parse/extract + clone analysis) speeds.

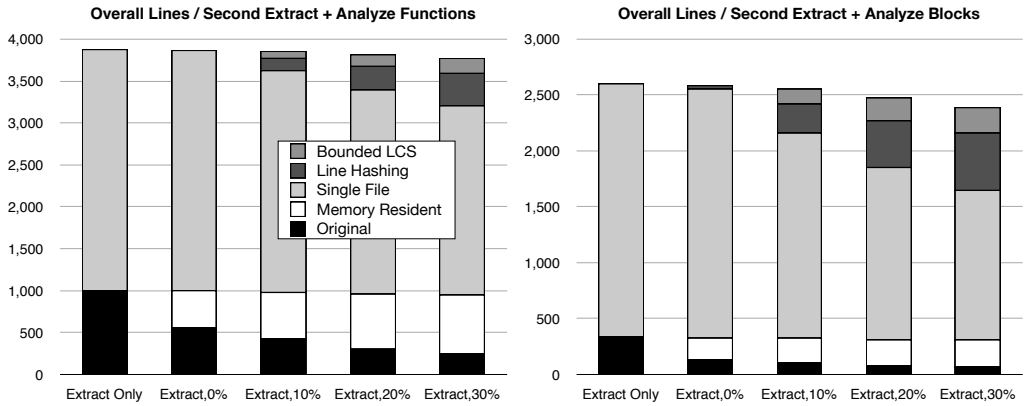

Figure 6: Change in overall NiCad processing speed, including both extract and clone analysis (lines per second), for function (left) and block clones (right). Note that we include extraction time for every near-miss threshold. In practice extraction need only be done once for all.

greatest effect at the function granularity. Not surprisingly, our improvements to the comparison algorithm itself, line hashing and LCS optimization, become increasingly important at finer granularity (blocks), and the overall improvement in processing speed is much greater (at a near-miss threshold of 30%, about 39 times faster for blocks compared to about 14 times for functions).

Clone Analysis Speed. Figure 7 shows the improvements in fragments per second clone analysis speed for our four test thresholds at each granularity. As we can see, by far the greatest improvement is at the 0% (exact clone) threshold. At first this was puzzling, since the exact clone detection algorithm should not have changed. However, on inspection we discovered that, for generality, the original NiCad prototype actually was using its LCS length function to test for exact clones, which of course the new implementation does not. The overall improvements in clone analysis speed are large – a factor of over 800 times faster for functions and 900 times for blocks at the 10% threshold for example. For the reason above, the improvement at the 0% (exact) threshold level is much larger than even those factors. In the near-miss cases, clearly the effect of our bounded LCS optimization is the dominant factor, accounting for about four fifths of the overall improvement at the 10% threshold at each granularity, and about two thirds at the 30% threshold.

Scalability. One of our original motivations for tuning NiCad was its inability to process really large systems in reasonable time, so while we can observe large speed increases, it is important to consider what effect our efforts have had on the processing of large systems. Table 8 shows our results for function clones in two

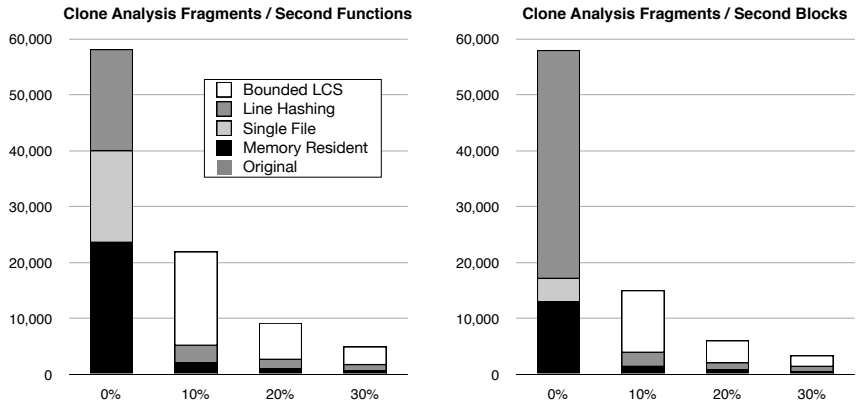

Figure 7: Change in NiCad clone analysis speed (fragments per second).

large systems – the Linux 2.6.24.2 kernel, and an artificial system consisting of all 48 source releases of the FreeBSD kernel from release 1.0 to the current 8.0 release. (The latter is being analyzed as part of a longitudinal study of functions in FreeBSD.) As we can see, the processing of the Linux kernel has improved enormously, down from 7 days to 33.5 minutes for function clones at the 30% near-miss threshold. For the amalgamated FreeBSD system, the times are much longer, at about 5.7 hours for the 30% threshold, which seems disproportionately slow compared to the Linux times until one realizes that such an amalgamated system is the worst case for our clone detection algorithm, since it has so many clones. This reduces the effect of the LCS length optimization, since it is aimed at comparisons that fail. Nevertheless, clearly our optimizations have taken NiCad from barely useable on large systems to practical for much larger systems.

Next Steps. Having tuned clone analysis, for most mid-sized systems the overall NiCad processing time is now dominated by the parse / extract stage, which is dependent on the TXL parser, which has already been tuned over many years. While there remain some opportunities for tuning the grammars to increase TXL parsing speed, two other strategies come to mind. First, since the extraction of potential clones from source files is independent for each source file, we can parallelize the extraction all the way down to the individual source file, and for example on our quad-core machine we can improve extraction speed by a factor of four simply by running four file parses at the same time, and even on a dual core laptop we can double the extraction speed. For future 8- and 16-core processors, this can clearly be a viable option. Alternatively, we can use a lighter weight algorithm, such as Cox's lexical approximation [24], to parse and extract potential clones.

Memory Issues. We have purposely not tried to estimate memory improve-

System	#Files	#Lines	#Fcns		Extract	0%	10%	20%	30%
Linux 2.6.24.2	9,714	463,198	155,556	**Old time**	58m	2.5 days*	3.5 days*	5 days*	7 days*
				New time	13m 23s	7.29s	4m 25s	15m 58s	33m 32s
		* 4 way parallel		**Clone pairs**		3,281	4,659	10,907	20,017
				Clone classes		932	1,674	3,977	6,735
FreeBSD 1.0 - 8.0	78,485	60,578,220	1,031,346	**Old time**	36h	2 weeks*	3 weeks*	4 weeks*	6 weeks*
(all 48 releases)				**New time**	9h	47m 3s	1h 27m	2h 49m	4h 41m
		* minimum estimate		**Clone pairs**		6,265,179	7,634,671	9,343,159	11,039,686
				Clone classes		108,428	89,642	76,273	68,510

Table 8: Original and new processing times for Linux and amalgamated FreeBSD releases.

ments in this work, instead limiting our new system to 2 Gb, the memory size of a typical modern laptop. In addition to running on our Intel Linux test machine, all of our runs, with the exception of the original NiCad, have been duplicated on a 2 Gb dual core Apple MacBook Pro to validate performance and memory use. The reason we have not statistically compared memory use is mainly because it is very difficult to measure, particularly for the original Perl-based NiCad rapid prototype, which uses multiple processes to perform the comparisons.

10. Related Work

While it is not the purpose of this paper to compare tools or processing speeds with other systems, it is important to set our results in a rough context. There are many fast clone detection tools, and we do not have room to review them all here. For a complete set see our recent survey of the state of the art [4].

Token-based systems such as CCFinder [25] and CPMiner [26] have been shown to be even faster than our tuned NiCad using token-based comparison for exact and renamed clones, handling relatively large systems in seconds by avoiding parsing. While their recall and scalability are unmatched, in general token- and text-based systems do not exhibit high precision without post-filtering, and also do not return structurally meaningful fragments without post-processing of results. Their effective overall performance can be much reduced by taking these extra costs into account. They also do not handle line-level near-miss clones. Metrics-based clone detectors such as Mayrand's [12], on the other hand, do well with near-miss clones and exhibit speeds similar to tuned NiCad. However, they can have even greater difficulties with precision.

In general, other successful parser-based systems, such as CloneDr [27] and the Bauhaus clone detector [11], use AST-based tree comparison to yield results that have been shown to be high precision, although sometimes at the cost of perfect recall. A lot of research has gone into tuning and optimizing them using

techniques such as suffix trees [11], and recently they have become very fast, processing large systems in a matter of seconds. Semantic Designs' CloneDr [27] handles a wide range of different languages and exploits parallelization [13] to achieve throughput and scalability to even the largest systems at high accuracy.

11. Implications for Other Tools, and Lessons Learned

As the scale of software systems grows from individual applications, to whole systems, to multi-version repositories and now to entire internet software corpora, the ability for analysis tools such as clone detectors to scale becomes increasingly important. Applications such as mining software repositories, reverse engineering large software systems, web service discovery, software evolution and migration to the semantic web all pose problems of scale involving processing and comparison of large numbers of individual source fragments or other artifacts for which our experience and observations can be brought to bear. Even if scale is not an issue, in smaller analysis applications the speed of source analysis can be a dominant consideration when interaction is involved, as for example for embedded analysis in an IDE, or when used in software maintenance for incremental analysis of changes for potential bugs.

Three particular applications of direct interest are the analysis of large bodies of software code for malware detection [28], in which source code must be searched for instances of patterns using normalization and similarity comparison in a way very much analogous to clone detection; analysis of commercial software code for copyright and licensing issues such as open-source license "contamination" [29], in which the code of a software system must be compared to the entire open source corpus (e.g., the eight distribution DVDs of Debian Linux [16]); and analysis of student assignment code for plagiarism of solutions from the web. Each of these problems involves both large numbers of small source files and large numbers of approximate comparisons which can benefit from both our file-reduction strategy and our use of the bounded LCS optimization. The results reported here can directly help guide implementers in how they can expect each of these strategies to be effective in scaling these source code analysis applications.

Our experience tuning NiCad has taught us a number of valuable lessons that can be used in tuning the performance of any tool. First, *remember to measure*. Our initial assumption with the original NiCad Perl implementation was that it was using machine resources as well as it could, and that our only choice was to use concurrency to speed up the clone analysis process. While introducing multi-processing did improve the performance of our original prototype tool by

a small factor, the change was nothing compared to the kinds of optimizations reported in this paper. By using system tools such as Linux's *time*, *top* and *ps* to observe and measure the actual use of machine resources, we were able to see that the majority of CPU time was actually being spent in system overhead related to file input/output and inter-process communication, which is only made worse by multi-processing.

Once we had moved to an in-memory comparison algorithm to reduce file input/output, measuring again using these same tools also allowed us to observe the large system overhead associated with large numbers of files, an operating-system dependency that we did not expect. This leads to our second lesson: *reduce system overhead first*. There is no point in optimizing your own algorithms if most of the time is taken up by system tasks associated with memory management, input/output and inter-process communication. These aspects must be addressed first, in our case by reducing the number of files and file accesses to a minimum, first using in-memory rather than file-level comparison, and second by using single XML-encoded large files to represent the thousands of small ones.

Once system overhead is brought to a minimum, which can be evaluated by measuring to see that at least 95% of CPU time is spent in "user mode" as measured by *time* or *top*, the third lesson is: *tune data representation to minimize repeated computations before tuning algorithms*. In our case, simply by changing to a hash-based line representation rather than raw text, we avoided repeated text comparisons and were able to speed up the comparison phase by a factor of three without any change to the algorithms.

After measuring to identify actual performance bottlenecks, optimizing file input/output to minimize system overhead, and tuning data representation to minimize repeated computations, the final step and last lesson is: *specialize general algorithms to the task*. In our case, because we were interested only in whether the length of the longest common subsequence between two sets of lines was greater than a certain threshold, we could look inside the LCS algorithm and optimize to cut off as soon as we knew that the difference was too large. Exposing the details of such standard algorithms or library routines is normally considered bad practice in software engineering, but when optimizing for performance, such specializing can have huge effects.

Finally, there is a tuning meta-lesson to be had from our experience: *don't parallelize until you've done everything else*. In the modern mutli-core world it is easy to believe that the best solution to tuning performance of research tools is concurrency. Our first attempt at tuning our research prototype involved both a significant effort to parallelize the process and the use of a larger, more expensive

multi-core computer to run it. While this strategy did allow us to run our first large experiments and publish our initial results, applying the lessons above led to a solution that was easier to implement, more scalable, hundreds of times faster, and required only the single core processor and 2 Gb memory of a standard laptop.

12. Conclusions and Future Work

In this paper we have shared our experience in tuning a naive research rapid prototype source analyzer, the NiCad clone detector, to scalable production speeds using a number of practical and technical optimizations. We have analyzed the effect of each of our changes and their relative importance to the overall improvement. Finally, we have outlined some lessons we have learned about performance tuning in general. It is our hope that others may be able to gain from our experience some hints on how they might improve the performance of their own program analysis systems and tools, and avoid the pitfalls of early parallelization.

Our work continues, and we have begun exploring optimization of the extraction phase by parser specialization (that is, by customizing the TXL parser and grammars to the extraction task), in a way analogous to the way we specialized the LCS algorithm for near-miss comparison. NiCad's potential clone extraction architecture lends itself very well to incremental clone detection [30], and we have recently adapted NiCad to an incremental version that can add new source files and update clone detection results in interactive real time, even for quite large systems (e.g., on the order of one second for *JHotDraw* and *httpd*). The result of our tunings has been released publicly as a research tool available to all [15, 31].

Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by an IBM Center for Advanced Studies faculty award.

References

[1] C. Kapser, M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Softw. Eng. 13 (6) (2008) 645–692.

[2] J. R. Cordy, Comprehending reality - practical barriers to industrial adoption of software maintenance automation, in: IWPC, 2003, pp. 196–206.

[3] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: ICSE, 2009, pp. 485–495.

[4] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Sci. Comput. Program. 74 (7) (2009) 470–495.

[5] C. K. Roy, J. R. Cordy, NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization, in: ICPC, 2008, pp. 172–181.

[6] C. K. Roy, J. R. Cordy, A mutation / injection-based automatic framework for evaluating code clone detection tools, in: Mutation, 2009, pp. 157–166.

[7] C. K. Roy, J. R. Cordy, Near-miss function clones in open source software: An empirical study, J. Softw. Maint. and Evol. 22 (3) (2010) 165–189.

[8] C. K. Roy, J. R. Cordy, Are scripting languages really different?, in: IWSC, 2010, pp. 17–24.

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Trans. Softw. Eng. 33 (9) (2007) 577–591.

[10] F. Franchetti, Keynote address: Towards automating black belt programming, in: iWAPT, 2011.

[11] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: WCRE, 2006, pp. 253–262.

[12] J. Mayrand, C. Leblanc, E. M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: ICSM, 1996, pp. 244–253.

[13] I. D. Baxter, Parallel support for source code analysis and modification, in: SCAM, 2002, pp. 3–14.

[14] C. K. Roy, J. R. Cordy, Scenario-based comparison of clone detection techniques, in: ICPC, 2008, pp. 153–162.

[15] J. R. Cordy, C. K. Roy, The NiCad clone detector, in: ICPC, 2011, pp. 219–220.

[16] J. R. Cordy, C. K. Roy, DebCheck: Efficient checking for open source clones in software systems, in: ICPC, 2011, pp. 217–218.

[17] J. R. Cordy, Exploring large-scale system similarity using incremental clone detection and live scatterplots, in: ICPC, 2011, pp. 151–160.

[18] J. R. Cordy, T. R. Dean, N. Synytskyy, Practical language-independent detection of near-miss clones, in: CASCON, 2004, pp. 1–12.

[19] J. R. Cordy, The TXL Source Transformation Language, Sci. Comput. Program. 61 (3) (2006) 190–210.

[20] R. C. Holt, J. R. Cordy, The Turing Plus report, Tech. Rep. CSRI-214, Univ. of Toronto (1988).

[21] N. Konz et al., Algorithm::Diff, *http://search.cpan.org/~nedkonz/Algorithm-Diff-1.15/* (2002).

[22] Longest common subsequence length, *http://en.wikipedia.org/wiki/Longest_common*

_subsequence_problem#Computing_the_length_of_the_LCS* (2010).

[23] D. S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.

[24] A. Cox, C. L. A. Clarke, Syntactic approximation using iterative lexical analysis, in: IWPC, 2003, pp. 154–163.

[25] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 654–670.

[26] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Trans. Softw. Eng. 32 (3) (2006) 176–192.

[27] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: ICSM, 1998, pp. 368–377.

[28] A. Walenstein, A. Lakhotia, The software similarity problem in malware analysis, in: Duplication, Redundancy, and Similarity in Software, Vol. 06301 of Dagstuhl Seminar Proceedings, 2006.

[29] D. M. Germán, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, Code siblings: Technical and legal implications of copying code between applications, in: MSR, 2009, pp. 81–90.

[30] N. Göde, R. Koschke, Incremental clone detection, in: CSMR, 2009, pp. 219–228.

[31] J. R. Cordy and C. K. Roy, NiCad download page, *http://www.txl.ca/nicaddownload.html* (2011).