Unique Pointers: Performance, Burden, and Inference

Mujtaba Ali
University of Maryland, College Park
mujtaba@cs.umd.edu

ABSTRACT

Cyclone is an actively developed, type-safe, C-like programming language. Historically, language designers have either leaned toward safety or toward explicit memory management. Cyclone aims to provide the safety of a language like Java, while providing the control over data representation and memory management of low-level languages like C. Cyclone features a garbage-collected heap. However, garbage collection is undesirable in some applications – the canonical example being the embedded space. To abate concerns with garbage collection, Cyclone has featured region-based memory management for well over a year. Until now, Cyclone regions were fairly coarse grained.

Developed by Dr. Michael Hicks of the University of Maryland, unique pointers are a new Cyclone construct. Unique pointers strive to regain the fine granularity of C's `malloc` and `free` without violating safety.

This paper details experiences with porting Cyclone programs to use unique pointers. Based on those experiences, this paper outlines a static, constraint-based analysis to help in the porting process. Additionally, this paper presents benchmarks comparing the performance of Cyclone programs before and after "uniquifying".

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms

Languages, Experimentation, Performance, Human Factors, Algorithms.

Keywords

Unique pointers, Cyclone, memory management, static analysis, constraint-based analysis, inference.

1. INTRODUCTION

Cyclone is a programming language embracing a unique (no pun intended) combination of type-safety and explicit control over data representation. Garbage collection is an easy way to enforce safety. However, garbage collectors often exhibit unpredictable behavior and often add unacceptable overhead. Regions were introduced into Cyclone as an alternative of sorts to garbage collection. Earlier, forms of Cyclone regions only supported allocation and deallocation *en masse*. Unique pointers, i.e., pointers pointing into the unique region, are a new Cyclone construct providing safe, individual object allocation and deallocation.

Ideally, a program could be ported such that all pointers exclusively point into the unique region. In such a case, the garbage collector need not be linked in the final executable – a great boon to the embedded space. Even in the presence of heap-allocated data that must be garbage collected, unique pointers can significantly decrease memory footprint.

A brief overview of unique pointers in Cyclone is provided in Section 2. If the reader is unfamiliar with general Cyclone constructs, she may consult the section entitled "Cyclone for C Programmers" of the Cyclone User's Manual [3].

The work presented in this paper consists of three main contributions:

1. Benchmark results of applications wholly or partially ported to use unique pointers are presented.
2. The programming burden of porting programs to use unique pointers is discussed.
3. A static analysis to partially ease the programming burden is detailed.

Cyclone was purposely modeled on C to ease porting of "legacy" C code. The porting process from C to Cyclone is the focus of an orthogonal effort and is not discussed in detail here. This paper is primarily concerned with comparing a "standard" Cyclone port of a program with a "uniquified" (still Cyclone) version of a program.

2. UNIQUE POINTER OVERVIEW

Put simply, a region is just a chunk of memory. Under Cyclone, pointers always point into a region. In other words, pointers' types are always annotated with a region. For example, pointers that live on the heap are said to point into the heap region and are annotated with the text `` `H ``. Individual deallocation is not allowed on the heap region; the heap region is always garbage collected. There are quite a few other regions useful under various circumstances (see Figure 1) [3].

The unique region permits individual allocation and deallocation. However, to statically (at compile-time) ensure safety, objects living in the unique region may have only one alias at any given time. Pointers are declared as pointing into the unique region by annotating their types with the text `` `U ``. For example, in the following declaration, `p` is a pointer pointing into the unique region; `p` is a unique pointer.

Region Variety	Allocation (objects)	Deallocation (what)	Deallocation (when)	Aliasing (objects)
Stack	static	whole region	exit of lexical scope	unrestricted
Lexical	dynamic			
Heap (`H)			automatic (GC)	
Unique (`U)		single objects	manual	restricted
Reference-counted (`RC)				

Figure 1. Summary of Cyclone regions.

```
int *`U p;
```

Below is a simple example of the type of error that can be caught at compile-time by Cyclone. The function `ufree` is analogous to C's `free` function.

```
int *`U p;
p = (int *`U) malloc (sizeof(int));
*p = 5;
ufree(p);  // "consumes" p
...
int q = *p + 7; // Error
```

Aliasing is not allowed on objects in the unique region.

```
int *`U p;
int *`H q;
p = (int *`U) malloc (sizeof(int));
*p = 5;
q = p; // Error
```

Working around the restriction on aliasing is undoubtedly the most difficult aspect of porting Cyclone programs to use unique pointers.

3. PERFORMANCE RESULTS

3.1 Overview

Generally, performance results are presented later in a paper. In this case, it makes sense to present performance results early. The reader must be convinced that unique pointers are beneficial; otherwise, discussions of programming burden and static analyses are futile.

A measurement of program memory footprint is presented here. Program execution time is also a significant measurement. For embedded environments, however, memory footprint can be as important as, if not more important than, execution time.

Ports of Boa – a high performance web server – and select modules from MiBench – an embedded benchmark suite [9] – were attempted. Boa was not successfully ported to use unique pointers, although it is very near completion. Footprint data was accumulated for the vanilla Cyclone version and the uniquified Cyclone version of the MiBench dijkstra module.

3.2 Memory Footprint

Some uniquified programs were not linked with the Cyclone garbage collector even though there is still heap-allocated data. This is because the original C versions of the programs did not deallocate this data and instead depended on the operating system to reap the data after the program process terminated. Such a technique is valid if data will live for the life of a program.

One example of such a uniquified program is the MiBench dijkstra module. Figure 6 shows the standard garbage collected run of the program. Figure 7 shows a run from the ported, unique pointer version of the program. The footprint benefits – 192KB versus 12KB (respectively) – are obvious.

3.3 Miscellaneous Details

If the Cyclone garbage collector was not linked in, the final executable was tested with Valgrind [4] to ensure there were no memory leaks. Valgrind is a run-time instrumentation framework, and an accompanying suite of tools, for debugging and profiling x86 Linux programs. The base Valgrind distribution includes Memcheck – a tool that detects memory-management problems such as memory leaks. Such testing reinforces that Cyclone's code generation is sound.

4. PROGRAMMING BURDEN

Even with performance benefits, programmers may not port their programs to use unique pointers if confronted with a heavy programming burden. Unfortunately, the programming burden is a significant drawback to uniquifying programs. Table 1 shows the number of lines that changed when porting. A greater LOC number does not necessarily mean that a greater number of changes will be required. Informally, the programming burden increases with regard to the complexity of data structures.

4.1 Manual Porting Process/Approach

Porting a standard Cyclone program to use unique pointers generally involved the following step:

1. Add a preprocessor directive to include `core.h`. Also, for convenience, open the `Core` namespace.

   ```
   #include <core.h>
   using Core;
   ```

   The `Core` namespace provides `ufree` and the `unique_region` handle. The reader can refer to [3] for information on Cyclone namespaces and region handles.

2. Look for calls to `free` and change these to `ufree`.

3. Find the declarations for the parameters to `ufree` and annotate these declarations to point into the unique region. For example, assume pointer `p` was a parameter to `ufree`, and p was declared as such:

   ```
   char *p;
   ```

   Then the declaration should be changed to:

   ```
   char *`U p;
   ```

   Alternatively, p may have been passed in as an argument to a function:

   ```
   void foo(char *p);
   ```

   In this case, the function parameter should be annotated:

   ```
   void foo(char *`U p);
   ```

   In the latter case, update any corresponding prototypes in `.h` files.

4. Compile the program.
5. Go through each compiler error and decide if an idiom (see below) can be used to resolve the error. If no idiom can be applied to resolve a specific error, that error must be resolved through thorough inspection of the source code.

4.2 Common Idioms

Here are many common idioms encountered when porting Cyclone program to use unique pointers. This list is not exhaustive by any means.

4.2.1 Idiom 1

Casts from calls to `malloc` (and other `alloc`'s) must be annotated.

```
p = (int *)malloc (sizeof(char));
        ⇓
p = (int *`U)malloc (sizeof(char));
```

Assume `p` is a unique pointer.

4.2.2 Idiom 2

If the left hand side of an assignment is a unique pointer, the right hand side must be changed to also point into the unique region. For example, assume `p` is a unique pointer:

```
p = q;
```

Then the declaration for `q` should be annotated with the unique region.

4.2.3 Idiom 3

Freeing global unique pointers is not allowed. This is because other code may be using the unique pointer (in the presence of concurrency) or a function earlier on the call chain may also refer to the global unique pointer after the free. As a side effect of separate compilation, the Cyclone compiler will not typecheck interprocedurally to determine if such a violation can occur. Instead, to preserve safety, the programmer should atomically swap a NULL value into the global unique pointer and then free it. If code subsequently dereferences the global unique pointer, a null exception will be thrown.

```
let temp_uptr = NULL;
temp_uptr :=: p; // atomic swap
ufree(temp_uptr);
```

Assume `p` is a global unique pointer.

4.2.4 Idiom 4

Many functions in the string library return pointers into the heap region. For example, `strdup` will accept an argument that points into any region and will return a new string allocated on the heap. Therefore, the following is invalid:

```
char *`U p = strdup("Infer me!"); // invalid
```

The Cyclone string library provides "region-aware" versions of many standard functions. These should be used instead. The region-aware functions will usually expect a region handle as the first argument. For example:

```
char *`U p =
       rstrdup(unique_region, "Infer me!");
```

Unfortunately, many of the region-aware function do not work with the unique region. As explained in [1], offending functions must be modified to accept the `TR` kind. A handful of functions – including `rrealloc`, `rstrdup`, and `rexpand` – were easily corrected during the course of this project.

4.2.5 Idiom 5

If a `return` statement returns a unique pointer, then (1) the return type of the associated function and (2) the declarations for all other returned pointers must be annotated with the unique region. Prototypes in `.h` files should be updated appropriately.

4.2.6 Idiom 6

Arithmetic is not allowed on unique pointers. Fat unique pointers are not exempt from this restriction. Unique pointers must always point to their base location so `ufree` can correctly deallocate memory. Fortunately, Cyclone provides an `alias` construct to temporarily alias unique pointers. Restricting aliasing to `alias` blocks helps Cyclone to statically guarantee that uniqueness properties are not violated. In particular, aliasing is convenient with loops; and `alias` blocks provide Cyclone programmers with a way to use unique pointers with the looping paradigm.

```
p = (char *@fat `U) malloc(sizeof(char)*5);
q = p;
for(i = 0; i < 5; i++)
    *q++ = 0;
       ⇓
p = (char *@fat `U) malloc(sizeof(char)*5);
{
let alias<`r> char *@fat `r q = p;
for(i = 0; i < 5; i++)
    *q++ = 0;
}
```

4.2.7 Idiom 7

If a unique pointer is passed as a function call argument, annotate the respective function parameter with the unique region. Prototypes in `.h` files should be updated appropriately. For example, assume `p` is unique in following statement:

```
foo(p, 5);
```

Then `foo` should be annotated like so:

```
void foo(char *p);
        ⇓
void foo(char *`U p);
```

4.2.8 Idiom 8

If a unique pointer is consumed via an assignment to another unique pointer, and later the consumed pointer is dereferenced, attempt to substitute the unique pointer that was assigned to in its place. Assume both `p` and `q` are unique in the following example.

```
p = q; // q consumed
foo(q->x);
        ⇓
p = q; // q consumed
foo(p->x);
```

4.2.9 Idiom 9

If a pointer has been previously annotated with the unique region and later it's discovered that the pointer's data structure is used in a cyclic fashion, then the pointer, and any `struct` fields involved in the cycle, must point into the heap region[1].

```
struct bar {int x; struct bar *`U next;};
void foo(struct bar *`U p) {
    p->next = p;
}
        ⇓
struct bar {int x; struct bar *`H next;};
void foo(struct bar *`H p) {
    p->next = p;
}
```

4.2.10 Idiom 10

If a pointer has been previously annotated with the unique region and later it's discovered that the pointer takes the address of a variable, then the pointer must point into the heap region. An alias creating in this manner is difficult for the typechecking flow analysis to track.

```
char *`U p;
p = &q;
        ⇓
char *`H p;
p = &q;
```

4.2.11 Idiom Frequency

As shown in Table 2, idiomatic changes comprise the vast majority of the total modifications while porting small programs to use unique pointers. Regardless, if even a few idioms could be automated via a static program analysis, a programmer undertaking a port will save considerable time and frustration. Furthermore, an automated analysis is less likely to introduce errors. Based on these idioms, one may instinctively sense that a set-constraint based analysis is a natural choice for automating the porting process. The author believes the manual porting process can be fairly termed "human constraint solving."

4.3 Weaknesses of Unique Pointers

List structures are a difficult paradigm to work with uniquely and some list function implementations require tedious workarounds [1]. The cause is essentially the "subtyping under references problem."

In general, the solution to subtyping under references is to cast the subtype as a constant so the subtype cannot mutate the reference [5]. Figure 2 shows the basic subtyping rule for Cyclone regions. If region `` `r1 `` outlives region `` `r2 ``, then `` `r1 `` pointers can be used as `` `r2 `` pointers. That is, pointers into the region `` `r1 `` are subtypes of pointers into the region `` `r2 ``. However, the polymorphic type variable `` `a `` is invariant. In other words, the rule presented in Figure 3 is invalid. Conversely, Figure 4 presents a valid rule that prevents an aliasing subtype from mutating its supertype's content.

$$\frac{\texttt{`r1} \le \texttt{`r2}}{\texttt{`a *`r1} \le \texttt{`a *`r2}}$$

Figure 2. Basic Cyclone subtyping rule.

$$\frac{\texttt{`r1} \le \texttt{`r2} \quad \texttt{`a} \le \texttt{`b}}{\texttt{`a *`r1} \le \texttt{`b *`r2}}$$

Figure 3. Invalid subtyping rule due to subtyping under references problem.

$$\frac{\texttt{`r1} \le \texttt{`r2} \quad \texttt{`a} \le \texttt{`b}}{\texttt{`a *const `r1} \le \texttt{`b *const `r2}}$$

Figure 4. Workaround for the subtyping under references problem.

4.3.1 List Idiom

Traversing through a list to retrieve or update a node's data is accomplished via a two-step process:

1. Creating a copy of the list structure data type, but with `const` pointers.
2. Cast the list to the `const` version before traversing.

The following code illustrates the problem lists pose in the context of unique pointers:

```
struct bar {int x; struct bar *`U next;};
void foo(struct bar *`U p) {
    struct bar *`U q;
    q = p->next; // consumes p->next !!
```

[1] Technically, a reference-counted pointer could be used. This paper, however, only considers unique and heap pointers.

```
}
```

Working around this problem requires the programmer to create a "clone" of the original list data structure, but with `const` pointers. The list can then be cast to a `const` version, after which it can be freely aliased. For example:

```
struct bar<`r>
{int x; struct bar *`r next;};
struct cbar<`r>
    {int x; struct cbar *const `r next;};
void foo(struct bar<`U> *`U p) {
    {
    let alias<`x> struct cbar<`x>
        *const `x q =
        (struct cbar<`U> *const `U) p;
    // freely alias p->next
   }
}
```

5. INFERENCE ANALYSIS

Intuitively, many of the idioms presented above are targets for automation. This section presents a sketch of a static analysis to infer unique region annotations. Additionally, the analysis will alert the programmer to locations in the source where a particular idiom may apply. The analysis need not be sound or complete. If the analysis can automate a large portion of mechanical changes, the analysis will be valuable even in the absence of soundness and completeness.

The analysis presented here is different from the typechecking algorithm the Cyclone compiler uses to guarantee uniqueness properties are not violated. Cyclone's typechecker uses a forward analysis while this analysis is a backward analysis. Furthermore, Cyclone's typechecking algorithm is intraprocedural – a common limitation in the presence of separate compilation. In contrast, the analysis presented here is interprocedural.

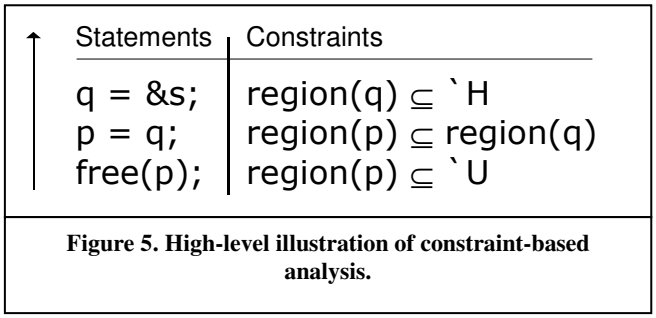

Figure 5. High-level illustration of constraint-based analysis.

5.1 Overview

The general analysis algorithm is quite simple. The analysis works backward through a program's CFG (control flow graph.) Only pointers that are `free`'ed are tracked by the analysis. All tracked pointers start as pointing into the unique region; all tracked pointers are "innocent until proven guilty." As the analysis works backwards through the CFG, constraints are generated on the region annotations of pointers encountered by the analysis. It could be that a constraint generated further up the CFG will ultimately force pointers into the heap region (see Figure 5[2].)

5.2 Constraint (and Alert) Generation

Constraint and alert[3] generation can be modeled with respect to the idioms presented in the previous section (see Table 3.) While perusing Table 3, the reader may find it useful to refer back to the original idiom descriptions. Assume an online constraint-solver.

5.3 Constraint Solving

The constraints generated using the patterns in the previous section can be solved using a standard constraint solver, such as the solver used by Andersen's alias analysis [7].

6. FUTURE WORK

Boa, at only a couple thousands of lines, can at best be considered a medium-sized program. Yet, porting Boa proved quite difficult and it is unclear if the process of porting programs to use unique pointers is scalable to larger programs. Further research will determine if the manual porting process and/or the automated, analysis-assisted porting process are scalable. However, this might not be an issue with Cyclone's embedded systems audience.

Program execution time was not measured during benchmarking. Furthermore, for Boa it would make sense to measure throughput using a load handling test tool.

Some programming paradigms are difficult to simulate with unique pointers. Lists in general, and especially circular lists, are problematic. However, reference-counted pointers [1] can be used with circular lists and future work can combine idioms for unique pointers with idioms for reference-counted pointers. Some list operations – e.g., copying – must currently be implemented recursively. It might be fruitful to investigate what is so different about recursion versus explicit iteration. Modeling explicit iteration with unique pointers will significantly increase programmer accessibility.

At present, the inference analysis relies on artifacts left over from when a C program was ported to Cyclone. That is, the analysis begins at `free` statements and works backwards. However, in Cyclone, `free` (not to be confused with `ufree`) is a noop; heap-allocated data is garbage collected. The analysis will most likely not work on a program written "from scratch " in Cyclone because `free` statements will not exist. A forward analysis would be ideal in that such an analysis could handle any Cyclone program. In the absence of `free` statements, a forward analysis could defer `ufree` insertion to the programmer. More interestingly, the analysis could try to infer where `ufree` statements should be placed.

Unfortunately, a working implementation of the analysis is not available at this time. Work has begun on an implementation using the CIL analysis and source transformation framework [6].

---

[2] The constraints generated here should not be confused with Cyclone *outlives* relationships [3]. For example, in Cyclone the unique region outlives the heap region: `` `H ≤ `U ``.

[3] Alerts are just messages printed out to the programmer's console.

The CIL framework comes with an implementation of Andersen's alias analysis, complete with online constraint solver.

7. RELATED WORK

The reader may refer to [1] for additional performance results and a description of work related to unique pointers in general. The author of this paper does not know of any prior work to categorize uniquifying idioms or to infer unique region annotations in Cyclone.

8. CONCLUSION

Unique pointers in Cyclone provide measurable benefits. However, programmers face a significant challenge when porting existing programs to use unique pointers. For some, such as the embedded systems community, the burden might be worth it. A static analysis that aids in the porting process may help adoption of unique pointers.

Further work may reveal that Cyclone's current unique pointer implementation is ineffectual and in need of revision or addendum. In such a case, future researchers can learn from drawbacks in the current implementation and/or try a modified approach.

9. ACKNOWLEDGMENTS

The author of this paper would like to thank the following individuals for their invaluable assistance: Dr. Michael Hicks, Dr. Jeffrey Foster, Jaime Spacco, Rob Sherwood, and Nikolaos Frangiadakis.

10. REFERENCES

[1] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. *Safe and flexible memory management in Cyclone*. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.

[2] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2002.

[3] Cyclone User's Manual. `http://www.research.att.com/projects/cyclone/online-manual/`.

[4] Julian Seward, Nick Nethercote, Jeremy Fitzhardinge. Valgrind. `http://valgrind.kde.org/`.

[5] Michael W. Hicks. Personal communication, 2003.

[6] George Necula, et al. CIL (C Intermediate Language). `http://manju.cs.berkeley.edu/cil/`.

[7] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, May 1994.

[8] Larry Doolittle and Jon Nelson. Boa webserver. `http://www.boa.org/`.

[9] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *4th Annual Workshop on Workload Characterization*, December 2001.

Appendix A – Tables and Figures

	Original LOC	Ported LOC	Diff LOC
bitcount	541	543	9
susan	1404	1406	9
dijkstra	268	277	27
patricia	282	288	22
stringsearch	3070	3073	10
boa	5088	5145	149

Table 1. Non-commented source code comparison.

	1	2	3	4	5	6	7	8	9	LI
bitcount										
susan	2									
dijkstra	1	1	1					1		1
patricia		2			1	2	1			
stringsearch	1		1	1						

Table 2. Idioms used while porting. LI stands for list idioms.

	Constraint	Alert
Idiom 1	`region(p)` ⊆ `region( (... *)malloc(...) )`	
Idiom 2	`region(p)` ⊆ `region(q)`	
Idiom 3		Swap global unique pointer with temporary pointer before `ufree`.
Idiom 4		Use region-aware version of string library function.
Idiom 5	`region(foo())` ⊆ `` `U `` {`region(ret`$_1$`)`, `region(ret`$_2$`)`,..., `region(ret`$_n$`)`} ⊆ `` `U `` where `ret`$_i$ `: i = 1 to n` are pointers returned by the foo.	
Idiom 6		Use `alias` construct.
Idiom 7	`region(p)` ⊆ `` `U ``	
Idiom 8		Substitute use of consumed pointer with new unique alias.
Idiom 9	`region(p)` ⊆ `` `H `` `region(field next of struct bar)` ⊆ `` `H ``	
Idiom 10	`region(p)` ⊆ `` `H ``	

Table 3. Constraint and alert generation rules.

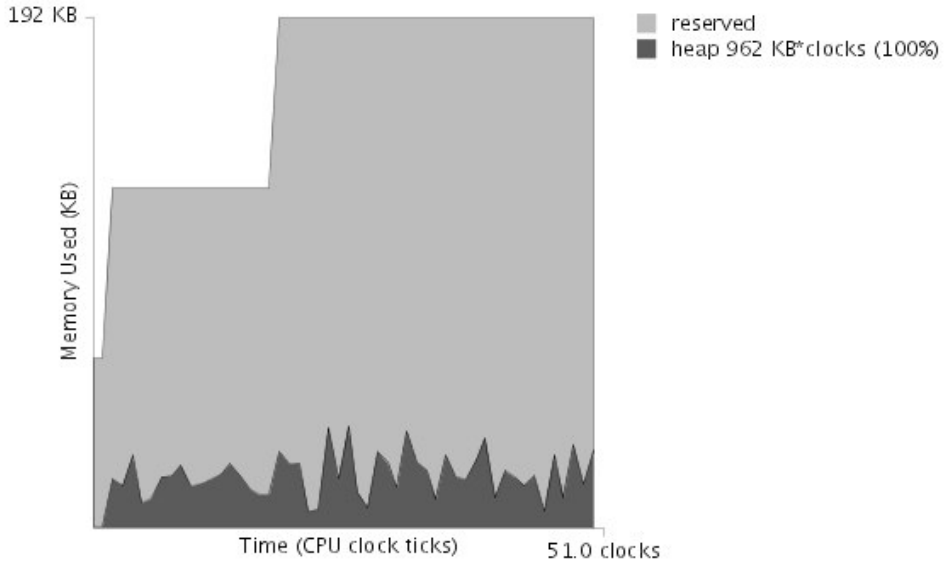

Figure 6. Memory footprint for dijkstra MiBench module. (standard)

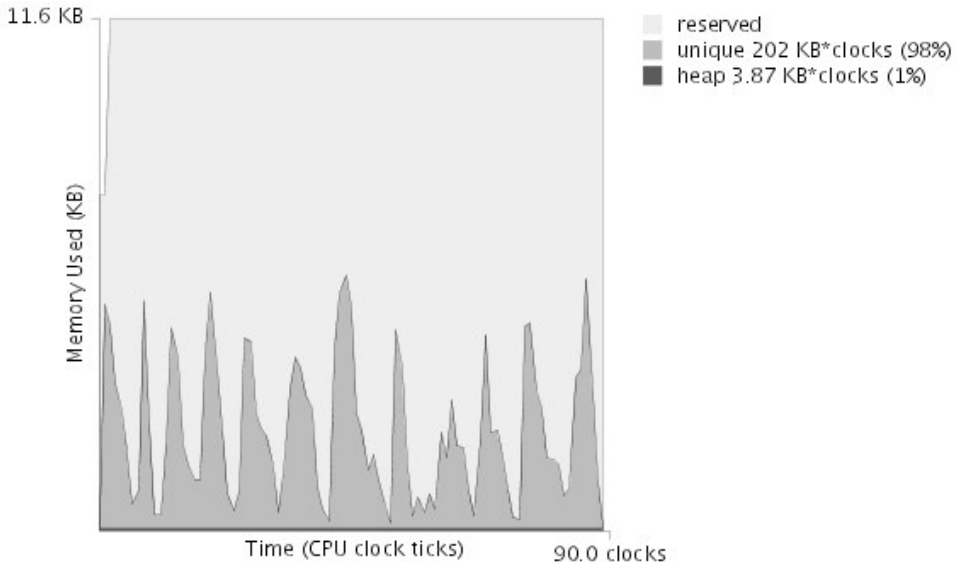

Figure 7. Memory footprint for dijkstra MiBench module. (unique)