Persistence is Hard, Then You Die!
or
Compiler and Runtime Support for a Persistent Common Lisp

J. H. Jacobs
M. R. Swanson
R. R. Kessler

UUCS-94-003

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

January 26, 1994

Abstract

Integrating persistence into an existing programming language is a serious undertaking. Preserving the essence of the existing language, adequately supporting persistence, and maintaining efficiency require low-level support from the compiler and runtime systems. Pervasive, low-level changes were made to a Lisp compiler and runtime system to introduce persistence. The result is an efficient language which is worthy of the name Persistent Lisp. [1]

[1]This research was sponsored by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1 Introduction

Integrating persistence (i.e. long-lived values) into an existing programming language is a formidable undertaking. Introducing persistence forces fundamental and far reaching changes into the persistent language implementation. The issues which necessitate these changes very often have no equivalent in the simpler world of volatile-value languages. So why even venture into the strange world of persistent programming?

Programming has always been about manipulating information and the volume of information has been steadily increasing. Computer aided design and expert systems all use data with life spans much longer than a couple of program executions (perhaps even outliving some programmers). Thus, constructing programs to manipulate long lived data is becoming an important task for software engineers. The tools available to programmers run from the I/O facilities of general purpose programming languages, through hybrid programs utilizing embedded database query language functions, on to persistent programming languages. Using general purpose programming languages requires the programmer to be intimately concerned with the low-level details of data access. Writing hybrid programs using a general purpose programming language to host queries in a database query language (e.g. embedded SQL [5]) allows the designer to escape some of the lower-level details of data access. However, data must be explicitly brought in from the store and explicitly saved; also the data must be translated between the representations of the database and the programming language. For general purpose problems, the solution is a persistent programming language. Only persistent programming languages provide a single, high-level tool for processing both volatile and persistent data.

1.1 Design Goals for Persistent Languages

Shoehorning a few persistent features into an existing language does not produce a persistent language, though it is a simpler and well travelled path. When introducing persistence into a language we believe there are three guiding principles: conforming to the definition and spirit of the base language, adequately supporting persistence and maintaining reasonable performance.

1.1.1 Conformity

The resulting language should have the look and feel of the original; otherwise a new language has been created, not a persistence-enhanced version of the original. Meeting this criteria requires that persistence be integrated into the language as transparently as possible. First, orthogonal persistence should be provided so that any data types a programmer would use in the base language are available in the persistent language. Second, the value management model of the language must guide the design of the persistent value management schema;

for example, languages with automatic storage management must support the automatic creation/destruction of persistent values. Third, the language must automatically detect the mutation of persistent values so that the changes will be properly persisted; relying on the programmer to explicitly flag all changes will lead to a lot of buggy programs.

1.1.2 Adequacy

Adequate support for persistent values must be provided. This includes atomic transactions, for maintaining the consistency of persistent values. Without them a properly written program cannot guarantee that only consistent sets of changes are allowed to persist, nor can multiple programs concurrently share persistent values. Since the persistent store might become quite large over time, a mechanism must be provided to allow a program to access reasonably-sized subsets of the stored values without requiring huge virtual address spaces. Lazy (demand) loading is one way to accomplish this. Failure to provide these will hamper the programmer trying to create well-crafted persistent programs.

1.1.3 Efficiency

Finally, the implementation must produce programs which are reasonably efficient and compact. If programs written in the persistent language are too slow or large, it may often benefit the programmer to use the original language and produce an *ad hoc* persistent program.

1.2 Existing Persistent Languages

Many well known programming languages have been enhanced with the ability to manipulate persistent data: Algol [4], C++ [9] [13] [1], Smalltalk [8], ML [11], and Lisp [12] [10] [2] [3] [6]. The C++ based implementations of persistence have been reasonably successful from a language design viewpoint. Their successful implementations have been due to the very explicit nature of value management in C++ . C++ variables are all strongly and statically typed; dynamically created values are explicitly allocated. C++ functions are second-class values and are not persisted, eliminating the difficult chore of persisting code. However, the language features which allow the easy introduction of persistence into C++ may also make the acceptance of persistent C++ more difficult. Many programmers attracted to the low-level of operations in C++ tend to pinch their bytes and cycles and are not willing to tolerate any extra resource consumption.

Languages which provide automatic storage management and other high-level features have a more difficult time introducing persistence. To conform to the spirit of automatic storage management, these languages must undertake automatic management of persistent values. In both ML and Lisp, the first-class nature of functions require that they too be persistable. The Lisp symbol data type is another source of difficulty because of the variety of roles it serves.

None of the Lisp systems mentioned above conform to the three principles. With the exception of LispO_2 [2] all of the systems forsake orthogonal persistence and limit persistence to an object data type; one even requires that updated values be explicitly marked by the programmer. LispO_2 provides a very limited transaction capability. Restricting the number of persistent data types, or weakening the transaction construct greatly simplifies the task of the language implementor, but dilutes the power or performance of the resulting persistent language.

1.3 UCL+P

UCL+P (Utah Common Lisp + Persistence) takes these problems "head-on" to solve all of the difficulties inherent in the three guiding principles presented above. A compilable, persistent Lisp was designed and then major low-level modifications were made to the UCL compiler and run time systems to efficiently support it. We believe that we have produced a language which is both persistent and still Common Lisp. We believe that our result is significant in and of itself, but also that our experiences will be helpful for others seeking to craft persistent languages.

The remainder of this article is broken into three parts. The first presents the design issues that necessitated modifications to the compiler, runtime system and data representations. The second part examines the changes made in those three areas. Finally, we conclude with a size and performance comparison between a corresponding set of persistent and volatile Lisp programs.

UCL+P is a large project and, while we would like to discuss all the interesting features, it is not possible in the space allowed. We will provide enough detail about the overall project so that the design and implementation can be put in context. However, we will not be discussing the design of the persistent store, nor the details of the Lisp interface to it, since they do not greatly affect the design and implementation of the language. Also missing will be a fuller discussion of writing and using UCL+P programs. We hope to focus on these areas in other forums.

2 Design of a Persistent Lisp

Before we can look at the compiler and runtime changes made to support UCL+P, we must first look at the design that resulted from applying the three principles defined above. The design of an efficient Persistent Lisp requires that we address several sometimes conflicting concerns. First and foremost, Persistent Lisp must produce programs that are fairly resource efficient, or else no one will use it. We must also preserve both Lisp semantics and the essence of the Lisp programming style; after all this is a Persistent *Lisp*. We provided UCL+P with very transparent persistent value manipulation features to accommodate the Lisp programming spirit. To support persistence itself, the atomic transaction was provided so that correct programs will be assured of leaving the persistent store in a consistent state, even in the event of system failure. In addition, since a set of persistent data can often be quite large, the language should permit a program to access as much persistent data as possible, within the constraints of the program's virtual address space. Finally, the symbol data type and the first class nature of functions require special attention. We will look at each of these issues in detail as we describe the design of UCL+P.

2.1 Practicality, Lispness, and Transactions

Persistent Lisp programs must make efficient use of both the CPU and memory if they are to be practical. To accomplish this, production Lisp programs need to be compiled and so the compiler must support persistence. In addition, the compiled code produced should be as fast as possible so that resident values, both volatile and persistent, can be manipulated at speeds comparable to volatile Lisp programs.

The software development process needs to be efficient as well. By conforming to Lisp semantics we make it easy for programmers to write persistent programs. The most important semantic issue raised by the introduction of persistence is the preservation of sharing since value sharing is pervasive in Lisp, occurring both intentionally and unintentionally. Sharing occurs whenever a value is referenced by two or more other values. Maintaining sharing semantics requires that all sharable data types be persistable (i.e. orthogonal persistence). Implementations which limit sharing to objects or another subset of sharable values cannot be faithful to the semantics of sharing.

Persistence introduces a semantic issue of its own: correct programs should always leave the store in a correct state. Since stored values will outlive the execution of the program an inconsistent result will have a very long lifetime (one might say it becomes a persistent problem). A mechanism is needed so that minor system failures do not permanently corrupt the store values. For example, an accounting program might need to transfer funds from one account to another. This involves two steps: decrementing the first account and incrementing the second account. If the store is to stay consistent, the updates to *both* accounts must be stored, or *neither*. We have adopted a database construct, the atomic transaction, which is used to collect a set of operations and make them all-or-nothing. When the transaction is completed the changes are committed (sent) to the store.

2.2 Transparency and Packages

A truly persistent language should provide transparent fetch and store of values so the programmer and the code need not be conscious of whether a value is persistent or volatile. UCL+P assures that all values are automatically present when needed. All mutated or created values are stored at commit time. The detection of values that need to be written back is performed implicitly by UCL+P; therfore, the programmer need not specifically mark a value as "dirty" which is akin to explicit storage management and can be a source of subtle and *insidious* bugs.

Requiring the programmer explicitly confer persistence on each value is also not compatible with the philosophy of automatic storage management. The method most compatible with Lisp is to dynamically confer persistence by reachability: all values (except symbols which are a special case) are initially volatile and become persistent when referenced by a persistent value. Without this approach, unusable values containing dangling references could be stored. Of course, some root values are required and Lisp symbols fill this function. In Common Lisp, symbols are associated with packages, a very simple container object. UCL+P extends the package facility to allow the user to define packages which are persistent; any symbols contained within a persistent package are persistent and any nonsymbolic values reachable from them are also persistent. Symbols are handled differently than other values. In Lisp, symbols allow for dynamic binding to values and functions. Conferring persistence by reachability onto symbols would constrain the late binding nature of symbols and would not be consistent with the spirit of Lisp.

The package mechanism also partitions the persistent values into semi-independent entities. A program can use any number of packages simultaneously and the packages may be shared concurrently with other programs. Even though the persistent values may be partitioned into separate packages, a single persistent package may be arbitrarily large, possibly larger than the virtual address space of the program. This would make the package useless, except that a program is unlikely to simultaneously (within a single transaction) access the entire contents of a large package. We have incorporated a lazy (demand) loading mechanism into UCL+P, thus enabling persistent programs to handle large packages.

2.3 Functions

Functions are first-class values in Lisp: they may be passed, stored, and evaluated. In addition, Lisp provides for a special type of function, the closure. Closures are the pairing of a function and a set of bindings. The first-class nature of functions requires that a persistent compiled Lisp be able to store and restore compiled code. Because the compilation and load process usually embeds information directly into machine instructions, persisting code is more difficult than saving the other data types.

3 Compiler and Runtime Support

The design of UCL+P required significant modifications to the compiler and runtime system. The representation of data values had to be fundamentally altered, which in turn, necessitated changes to the code generator and the runtime system. The need to persist compiled code also required changes to the compiler.

3.1 Data Representation Changes

The data area of UCL (original version) is broken up into three areas: the symbol table, the heap and the function area. Most values reside in the heap which is garbage collected as needed. Heap resident values are accessed via pointers contained in the symbol table, other values, stack frames, or function bodies. UCL+P adds two more data areas: the persistent heap and the persistent symbol table. All UCL+P values begin their lifetimes as volatile, heap-resident values. When values become persistent they must be moved out of the volatile heap and onto the persistent heap when the transaction completes. This is necessary so that newly persistent values cannot be accessed when the program is not inside the scope of a transaction; accessing mutable persistent values outside the body of a transaction would violate the conditions necessary for correct transaction semantics. When the program is not executing within the body of a transaction, the persistent heap and symbol table are protected against access via operating system memory protection facilities (e.g. MMAP on BSD Unix). O/S facilities are also used to implement lazy loading: the runtime system uses page faults on persistent heap accesses to trigger the loading of needed persistent values.

Because the newly persistent value might be referenced by other values, the relocation of the value must not leave any existing references dangling. To support this, the direct pointer reference mechanism was replaced with an indirect pointer reference (i.e. pointer to a pointer). Reference slots in values now contain pointers to an entry in the Indirection Vector (IV). The IV contains an entry for each heap resident value. Each IV entry contains access flags and the heap address of the value. With the IV mechanism a value can be safely relocated by changing its heap address in the IV. Without this, the entire data area would have to be searched so that all relevant pointers could be adjusted to follow the newly persistent value, an operation equivalent in cost to performing a garbage collection. Another approach would have been to keep the direct reference mechanism and perform a garbage collection at the end of each transaction. The choice between the two methods is a tradeoff between the time required to perform a garbage collection and the impacts of adding the IV into the system. We selected our approach because it supports smaller grained transactions than the "commit and garbage collect" approach.

Access detection is necessary to implement transaction semantics. The runtime system must detect and record read and write accesses to all persistent values so that this information can be passed on to the transaction validator at commit time. Part of the IV entry contains a read flag and a write flag which are set if the value is read or written. When the transaction commits, the IV and persistent symbol table are scanned to construct the access sets for the transaction. An alternative implementation would be to make the access flags a part of the values themselves, but then it would be necessary to scan the entire heap for updated values, which would be less efficient.

When symbols are created, they are permanently contained in a single package. Therefore symbols never change from volatile to persistent, so we can to use direct pointers to access them. Because named functions are accessed through the function slot of the naming symbol, using another level of indirection to get to the symbol would inevitably slow down the function calling process. The symbol table already held slots for the attributes of a symbol: value, function, property list, print-name, and package. Because symbols are directly accessed a slot was added to the symbol itself to hold the access flags.

Function references still use direct pointers, preserving the efficiency of function calling. Unlike other values, functions do not need to be relocated when the function completes. This can be done while preserving transaction semantics because functions are immutable and so do not produce inconsistent results when they are called outside a transaction. When a function with state, a closure, is made persistent, its data part is relocated into the persistent heap, but its code part is not.

3.2 Changes to Code Generation

Three major changes were made to the code generation portion of the UCL compiler. The instructions generated by the compiler needed to support both the extra level of indirection introduced by the IV and the access marking needed for transactions. The other change was to enhance the compiler to produce position independent code when compiling application functions.

3.2.1 Supporting Indirection and Access Detection

The UCL compiler uses a set of opencodes. These are effectively macros that expand into assembly code during the code generation process. Among these opencodes are the instructions for low-level Lisp primitive functions such as *car*, *cdr*, array access, etc. All of the opencodes that access heap resident entities needed to be changed to use the IV and to record accesses. The symbol accessing opencodes were modified to incorporate access marking. The changes were done in two steps. First the opencodes were modified to use the IV and in the second step the opencodes were changed to set the appropriate read or write bit for the value.

The UCL+P prototype system currently outputs 680x0 machine code. The 680x0 is a CISC style processor with a large set of addressing modes. Most of the opencodes required the addition of two instructions. The first instruction sets the appropriate read or write bit associated with the value or symbol. The second instruction dereferences the first pointer and leaves the heap address available for the rest of the opencode. Figure 1 shows the UCL and UCL+P opencodes for *car*. If a RISC style processor had been the compiler target it would have taken more instructions per opencode to support persistence, but the opencodes themselves would also be longer, so the relative increase in opencode size should be comparable to the CISC case.

UCL

```
moveal d1,a3            ; move first arg to address register
movel  a3@(-1),d1       ; move CAR value to result register
```

UCL+P

```
moveal d1,a3            ; move first arg to address register
moveb  #1,a3@(-5)       ; set read access flag in IV
moveal a3@(-1),a3       ; get heap pointer
movel  a3@(-1),d1       ; move CAR value to result register
```

Figure 1: UCL and UCL+P opencodes for *car*.

3.2.2 Persistable Compiled Code

Because UCL+P needs to store compiled code, it was necessary to modify the compiler to optionally produce position independent code (PIC). PIC output has been prototyped on the UCL compiler and we are in the process of integrating the PIC into the persistent compiler.

In our new PIC scheme a header was added to the compiled code for a function. The header contains constants referenced by the function and references to symbols. By placing the constants and symbol references in a linkage table, the task of storing and reloading compiled functions became feasible. The cost for this capability is an increase in code size and a decrease in performance. The performance decrease is due to the extra indirection required for accessing symbols; a non-PIC symbol reference would use an immediate instruction to access the needed slot while the PIC code must indirect through the function header. Fortunately, only application functions will need to be compiled with PIC; most runtime system functions already "persist" as part of Lisp and need not be made storable.

3.3 Runtime System Modifications

With the changes to the data representation, both the allocation routines and the garbage collector had to be enhanced. The allocators now have to provide both an IV entry for each new value created, as well as heap space. If either is unavailable, the garbage collector needs to be activated.

Adding the IV mechanism to UCL required that the garbage collector be modified. Since some Lisp programs can spend a considerable amount of time, perhaps as much as one third [14], performing garbage collection, changes to the garbage collector can greatly impact program performance. UCL uses a two-space, copying garbage collector; the collector copies live values from the current half-heap to the spare half-heap. In UCL+P the collector also reclaims dead IV entries. Since collection can be triggered by exhausting the IV table, the collector must perform the copying without consuming *any* IV entries. This required considerable care. Additionally, the collector must not leave any trace of its actions on the access flags of the values. If the collector were to leave any "fingerprints" on the values, the transaction commit mechanism would be forced to deal with a large number of false accesses.

4 Size and Performance of UCL+P programs.

After looking at the low-level modifications used to introduce persistence, we can look at how they affected the programs produced by UCL+P. We compared the performance of programs produced by UCL and UCL+P. The same source code was compiled by both compilers and executed using the corresponding runtime system. The test programs did not actually use the persistence mechanism, since we wanted to focus on the effects of the low-level changes and not on the performance of the backing store.

4.1 Code Size

As described above, some of the opencodes used for code generation had two instructions added to them. In the worst cases, this doubled the number of instructions. However, only a third of the opencodes needed to be changed. To get a feel for how the modified opencodes affect the size of the compiler output we compiled and loaded a large Lisp program (4400 lines) using the UCL and UCL+P compilers. UCL produced 155KB of application code while UCL+P output 170KB, a 9% increase. The prototype version of the UCL PIC compiler increases the code size by 7% and is likely to have a similar effect when integrated with the UCL+P compiler.

4.2 Data Size Impacts

The changes made to the data representation left the sizes of all stored values unchanged. However, the data size has been increased by the introduction of the indirection vector. Although each entry in the IV is very small (8 bytes), one entry is required for every heap resident value. The smallest heap allocated value in UCL is the *cons* cell, which takes 8 bytes, while vectors, matrices and strings can be very large. By examining the heaps of the runtime system, with and without the compilation routines, we get an average value size of about 20 bytes. (The average value size depends heavily on the relative mix of data types used in the program.) Based on this average value size and the fact that each IV entry takes 8 bytes, the IV size should be about 40% of the half heap size (active allocation occurs in only one half of the heap at a time). Therefore, the addition of the IV increases the storage needed for heap resident values by about 20%.

Besides increasing the size of the program, the IV also alters the data reference patterns. The IV entry must be touched before the value can be accessed which reduces the data reference locality. Although we have not directly measured the effect, the loss of locality likely impacts both virtual memory and cache performance.

4.3 Performance

As expected, adding support for transparent persistence slows down program execution. To measure the change in code performance we utilized the Gabriel benchmark set [7] which is tailored for Lisp programs; the results are shown in Table 1. When the tests were run, the runtime system used an IV that was 100% of the half heap size. The table also reports the average of the CPU-time ratios over all tests (Gabriel did not define a single metric). One disadvantage with the Gabriel benchmark set is its lack of programs that a modern Lisp programming style might produce. Newer programs would make fairly extensive use of structures and objects and would be less list intensive. We will extend the benchmark set to include these programming styles.

The CPU-time ratios range from 1.0 and 1.8. Programs which make extensive use of lists (e.g. Boyer, Browse) suffer from the higher slowdowns. If the persistent and volatile opencodes for a representative list operation, *car*, are examined (Figure 1) it is no surprise that list intensive programs are the most affected by the compiler changes. Integer and array intensive programs such as Puzzle and Triangle are the least affected by the changes. Floating point intensive tests such as FFT and Frpoly are between the two extremes.

Fortunately, further performance optimizations are possible. For floating point operations it should be possible to avoid marking them and bypass the IV since floating point numbers are immutable. For most other values, whenever a function accesses a single value repeatedly, redundant pointer dereferences and access marking can be eliminated.

Our results do not include PIC coded routines. When PIC code is used throughout the UCL runtime system and the application code, performance slowed down by about 10%. However, there is no need for most runtime system functions to be PIC because they are always present as part of the runtime system. Since 75-90% [14] of functions called are runtime functions, very little of the PIC slowdown should show up in overall program performance.

5 Conclusions

Producing a transparently persistent Lisp required fundamental changes to the UCL compiler and runtime system. Any system which attempts to introduce persistence into Lisp without resorting to such low-level changes must either sacrifice language semantics, or suffer severe performance penalties. After the changes were made the performance of the resulting system showed that 20% more space was needed for storing data and 9% more space was required, overall, for storing application code. Program performance also suffered with the language enhancement. Volatile-value-only programs saw CPU times increase by about 18% for integer intensive programs to around 43% for list intensive benchmarks, though we have identified some future enhancements to the compiler's optimizer which should reduce the slowdown.

We have produced a Persistent Common Lisp which adheres to the three principles defined in the introduction. While the efficiency of UCL+P is currently less than we wanted, the language conformity and adequacy of persistence goals have been unequivocably been met. As it stands, the transparent integration of powerful persistent value support makes UCL+P an optimal solution for constructing persistent programs using Common Lisp though it is unlikely to become a general purpose replacement for UCL.

References

[1] R. Agrawal and Gehani N. H. ODE (Object Database and Environment): The language and data model. In *Proc. Int'l. Conf. on Management of Data*, pages 36–45, Portland, Oregon, May-June 1989. ACM-SIGMOD.

[2] Gilles Barbedette. LispO_2: A persistent object-oriented LISP. In F. Bancilhon, C. Delobel, and P. Kanellakkis, editors, *Building an Object-Oriented Database System: The Story of O_2*, chapter 10, pages 215–233. Morgan Kaufmann, 1992. Also in Proceeding of the 2nd EDBT.

[3] P. Broadbery and Burdorf C. Applications of Telos. *Lisp and Symbolic Computation*, 6(1/2):139–158, August 1993.

[4] W. P. Cockshott. *PS-ALGOL Implementations: Applications in Persistent Object-oriented Programming.* Ellis Horwood, 1990.

[5] C. J. Date. *An Introduction to Database Systems, Volume I, Fifth Edition.* Addison Wesley, 1990.

[6] S. Ford, J. Joseph, Langworthy D., D. Lively, G. Pathak, E. Perez, R. Peterson, D. Sparacin, S. Thatte, Wells D., and S. Agarwala. Zeitgeist: Database support for object-oriented programming. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*. Springer-Verlag, 1988.

[7] R. P. Gabriel. *Perfomance and Evaluation of Lisp Systems*. MIT Press, 1985.

[8] A. L. Hosking, J. E. B. Moss, and C. Bliss. Design of an object faulting persistent Smalltalk. Technical report, Univerity of Massachusetts, 1990. UM-CS-1990-045.

[9] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, Oct 1991.

[10] Arthur H. Lee. *The Persistent Object System MetaStore: Persistence via Metaprogramming*. PhD thesis, University of Utah, Aug 1992.

[11] S. M. Nettles and Wing J. M. Persistence + undoability = transactions. In *Proceedings of the Hawaii International Conference on Systems Science 25*, 1992. See also tech-report CMU-CS-91-173.

[12] A. Paepcke. PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 1988.

[13] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989. Tech Report 824.

[14] Robert A. Shaw. *Empirical Analysis of A Lisp System*. PhD thesis, Stanford University, February 1988.

Name	With IV	With IV & Marking
Boyer	1.3035	1.4261
Browse	1.3720	1.5750
CTAK	1.0738	1.4180
Dderiv	1.4958	1.5805
Deriv	1.3333	1.4932
Destructive	1.2343	1.3829
Div-iter	1.5854	1.8780
Div-rec	1.4082	1.6327
Fact 1000	1.0698	1.1358
FFT	1.3750	1.4632
Fprint	1.1203	1.3354
Fread	1.1012	1.1994
Frpoly Power=2 r=x+y+z+1	1.0000	1.0000
Frpoly Power=2 r2=1000r	1.5000	1.5000
Frpoly Power=5 r=x+y+z+1	1.0000	1.5000
Frpoly Power=5 r2=1000r	1.1071	1.1429
Frpoly Power=5 r3=r in flonums	1.2500	1.5000
Frpoly Power=10 r=x+y+z+1	1.1905	1.4286
Frpoly Power=10 r2=1000r	1.1096	1.1790
Frpoly Power=10 r3=r in flonums	1.3023	1.4419
Frpoly Power=15 r=x+y+z+1	1.1673	1.3271
Frpoly Power=15 r2=1000r	1.1084	1.1686
Frpoly Power=15 r3=r in flonums	1.2715	1.3952
Puzzle	1.0855	1.1765
STAK	0.9903	1.1456
TAK	1.0000	1.0000
TAKL	1.1500	1.3833
TAKR	0.9615	1.0000
Tprint	1.1842	1.3333
Traverse-init	1.1250	1.2436
Traverse	1.5263	1.7368
Triangle	1.0202	1.2245
Average of Ratios	1.0866	1.2378

Table 1: Results of running the Gabriel benchmarks. Shown are the ratios of CPU time used relative to UCL program after adding indirection vector and after the addition of both IV and access marking.