Performance Visualizations using XML Representations

Yijun Yu* Kristof Beyls† Erik H. D'Hollander†

*CS, University of Toronto. 40 St George Street, M5S2E4 Toronto, Canada

†ELIS, University of Ghent. 41, 9000 Ghent, Belgium

Abstract

The intermediate representation (IR) forms the information exchanged among different passes of program compilation. The intermediate format proposed for extensibility and persistence is written in XML. In this way, the program transformations that were internal to the compiler become visible. The hierarchical structure of XML makes a natural representation for the abstract syntax tree (AST). A compiler can parse the program source into an IR, then output it as an XML document. Separated by orthogonal namespaces, other IRs are also presented in the same XML document, gathering program information such as dependence vectors, transforming matrices, iteration spaces dependence graphs and cache reuse distances. This XML document can be exchanged between the compiler and program visualizers for parallelism and locality. *Keywords*—

intermediate representations, XML, performance visualizations

1 Introduction

Designers of compiler systems have employed the intermediate representation (IR) to systematically define the data interface between its different subsystems [1, 6, 4]. An IR includes the most common data structures used in the compiler. The high level data structures include the abstract syntax tree, the symbol and the type table, the call graph and the dependence graph, and the low level data structures include the task graph and the instruction tuples.

In order to exchange information between different passes and to support flexible combination of compile options, the IR is made persistent through an intermediate format like `SUIF` in the SUIF compiler [15], and `Lcode` in the IMPACT compiler [7]. Similar efforts are also made to serialization of Java data structure through the `java.io.Serializable` interface [10].

Earlier efforts like ASDL [14] has been made to pickle or marshal the tree-like IR defined within individual compiler like SUIF. After writing several hundred lines of ASDL definition, thousands line of code implementing the API for pickling are generated. The API supports the sharing of information among different compilers or between a compiler and another programming tool.

Experience with ASDL in `lcc` [8] has shown how to retrofit an existing compiler by separating its C parser and the multi-platform code generator into two components communicating only through ASDL. Afterwards new optimizer components could be inserted without changing the two parts.

Both the designers and the users of ASDL have notified that the drawback of using ASDL is a duplication of data structure in memory [14, 8]. To improve the efficiency, the Aterm [13] targeting at minimize the storage of trees in more efficient binary formats.

With the advent of XML [3] as an information exchange standard, almost every corner of the software industry is inspired to share the once-incompatible information to the other XML enabled applications. This trends already deeply touched the compiler industry. Any XML document, in the first place, has to be valid against its DTD(document type definition) by an XML parser. Another example is the `GCCXML` Kitware project, where XML is adopted to output procedural information [9]. Both the Java serialization and the ASDL pickle can further be converted from and to the XML format [8].

However small, both ASDL and Aterm still have to occupy memory as additional overhead when retrofit an existing compiler, while pure event-based XML parser `SAXparser` does not. In this paper, we propose a thorough use of XML in the compiler infrastructure. That is, to exchange all the IR data structures in XML as needed. This will support persistence of the compiler internal information and make it extensible for outside tools to share the information with the compiler. As will be shown in the paper, not only tree-like AST can be represented, various IR data structures including complex dependence graphs can be expressed in XML. The use of orthogonal namespaces for various data structures makes it possible to bind related information in a single XML document. The apparent huge requirement for XML storage can be alleviated by a stand-alone compression tool.

The remainder of the paper is given as follows: section 2 discusses the reason to XMLize the common IR used in most of the compilers; section 3 overviews the compiler system combined with external transforming tools and vi-

Table 1: The name-spaces for our compiler IR

namespace	representation
ast	an abstract syntax tree(AST)
yaxx	a YACC parsing tree
par	an identified parallel or sequential loop
isdg	an iteration space dependence graph
hotspot	performance bottleneck locations
trace	an execution trace of memory instructions
cache	parameters for simulating a cache
rdv	a reuse distance vector

sualizers; section 4 visualize other program information like loop dependence and cache behavior by sharing information between the compiler and various visualizers; section 5 concludes the work with future perspectives.

2 Representations in XML

An XML document for the intermediate representation (IR) is a tree of XML tags. In our design, an IR document has the following form:

```
<IR phase="parsed" source="gaussjordan"
 xmlns:ast="http://elis.rug.ac.be/fpt"
 xmlns:par="http://elis.rug.ac.be/par"
 xmlns:yaxx="http://elis.rug.ac.be/yaxx"
 ...>
...
</IR>
```

The root ”IR” tag of the document tree has two basic attributes: ”phase” indicates which compiling or visualizing phase the IR belongs to; ”source” indicates the name of the source program under investigation. As the document root, it carries a number of uniform resource identifiers for orthogonal name-spaces, e.g. see table 1. Though they are not a complete set of IR used in program transformations and visualizations, the extensibility of XML opens possibility for name-spaces of additional applications.

Each child XML tag is prefixed with the name-space corresponding to a certain intermediate representation. For example, the abstract syntax tree has an `ast:` prefix, the parsing tree has a `yaxx:` prefix, the parallelizable loop has a `par:` prefix, etc.

Global data structures are put as the immediate subtrees of the document root. E.g. `ast:N_PROGRAM` as the root of an AST; the name of the starting rule `yaxx:executable_program` as the root of a parsing tree, a `trace:sequence` as the root of a program execution trace, etc.

Local data structures are put as the children of an AST node corresponding to its scope. For example, the dependence test annotates each parallel DO loop ”ast:S_DO” with the parallel information:

```
<ast:S_DO> <par:true/> ... </ast:S_DO>
```

Similarly, tag `isdg:graph` is a child of the outermost loop of a loop nest `<ast:S_DO>`, denoting an iteration space dependence graph. Other information, such as `hotspot:histogram` which contains statistic information for cache miss patterns can be attached to any statement in the AST, either a program unit, a loop nest or even a single memory instruction. In this way, program semantic information are othogonally associated with the AST in one XML document.

3 Extending the system architecture

Using XML to represent the IR in a compiler, we aim to extend the compiler with the ability to exchange information with external tools, such as Omega calculator [11] and program visualizers including ISV [17] and CacheVis [16]. The design of the integrated system is shown in figure 1. As a source-to-source optimizing compiler, FPT is separated into several phases: the first phase parse a source Fortran program into AST; then each phase either changes the AST or creates additional IR; the last phase unparse the AST back to a target Fortran program.

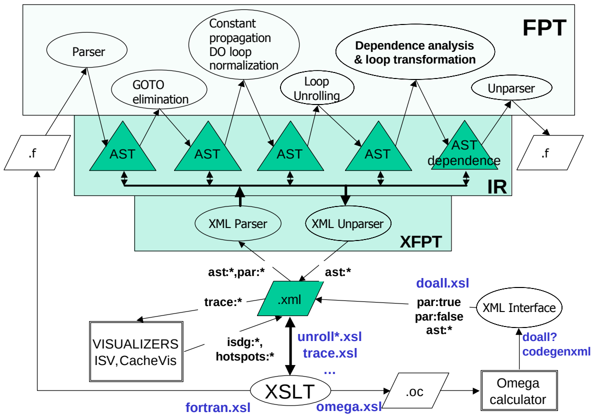


Figure 1: The extended system architecture.

The separation of phases was internal to FPT before introducing the XML interface: XML parser and XML unparser. They bridge between the IR and the XML document, which is then given to external tools, namely, XSLT, Omega and Visualizers.

The XSLT is a collection of XML transformers: `unroll*.xsl` performs loop unrolling transformation which has been implemented in FPT. Now it is performed purely in XSLT, showing the potential to implement a compiler transformation outside the compiler; `fortran.xsl` just produces the Fortran source code from the syntax information in the IR document, which is similar to FPT unparser. But it does not require the document to be loaded into FPT and does provide a convenient debugging tool for checking the correctness of transformations; `omega.xsl` extracts the iteration space and dependence equations from an AST as input to the Omega calculator, and `doall.xsl` interfaces with Omega calculator to annotate the parallel loop loops in the AST; finally, `trace.xsl` creates an instrumented program that allows dependence analyzers in ISV and cache simulator in CacheVis to visualize the dependence graph and cache miss hot spots.

4 Program Visualizations

The AST in XML form can be obtained from the XML-enabled parser such as YAXX, then visualized either by using commercial XML editors such as `XMLSpy` or open-source graph layout tools such as `GraphViz` [5] and `Vcg` [12]. Besides the AST, other semantics information can also be visualized.

For performance optimization, two kinds of program characteristics are considered extremely important. One is the parallelism, the other data locality. The semantics information for studying parallelism is data dependence, while the information for studying the cache behavior is reuse distances [2] in cache stack history. The data dependences can be visualized using a loop dependence visualizer ISV [17]; the data locality can be visualized using a cache visualizing tool CacheVis [16]. The integration glue for them with the compiler is naturally XML.

4.1 AST in XML for visualization

Abstract syntax tree(AST) is the most commonly used IR for the program syntax. The syntax of high-level programming languages are expressed in production rules in Backus Naur Form (BNF). On reduction of a production rule, the right-hand sided terms become the children of the left-hand sided term in a syntax tree.

We adapted `bison`, an open-source variant of YACC, to produce a DTD from the YACC grammar and a parsing tree in the DTD-compliant XML corresponding to a program follows the grammar. The "start" non-terminal becomes the root of the XML document. A terminal node is always a leaf node in the XML tree. A non-terminal with non-empty production rule have the RHS nodes as its children.

A parsing tree might be used as an AST before several concerns are addressed.

- The parsing tree from YACC grammar are usually too deep because a repetitive structure like a list is often represented by recursive production rules:

  ```
  List : Item | Item List
  ```

  However, we can make it more readable by simplifying the production rule using a feature of document type definition(DTD) element:

  ```
  <!ELEMENT List ((Item)+) >
  ```

- Secondly it is redundant to have two rules like "A:==B, B:==C" while the first rule is the only rule defining A. It can be removed by replacing every occurrence of A into B.

After the above simplifications, a grammar does not lose its expressiveness. To express a Fortran program, its AST is expressed by the following document type definition:

```
<?xml version="1.0" ?>
<!ELEMENT N_PROGRAM
((N_COMMENT|N_PARAM|N_TYPE|N_STATEMENT|...)+ )>
<!ELEMENT N_COMMENT (#PCDATA)>
<!ELEMENT N_PARAM ((N_PARA)+)>
<!ELEMENT N_TYPE (type (N_ARRAY_DECL|N_VAR)+)>
<!ELEMENT N_STATEMENT
    ((S_DO | S_IF | S_ASSIGN | ... )+) >
...
```

The XML tags correspond to different types of AST node.

Consider a small Fortran program as follows.

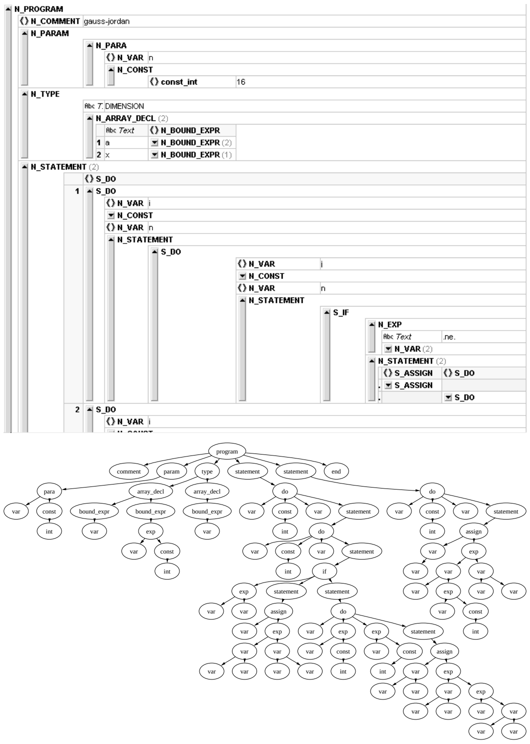
```
!gauss-jordan
parameter(n=16)
dimension a(n,n+1),x(n)
do i=1,n
   do j=1,n
    if(i.ne.j) then
     f=a(j,i)/a(i,i)
     do k=i+1,n+1
      a(j,k)=a(j,k)-f*a(i,k)
     enddo
    endif
 enddo
enddo
do i=1,n
 x(i)=a(i,n+1)/a(i,i)
enddo
end
```

Its AST is generated as an XML document shown in XMLSpy (Figure 2a). The AST can also be transformed by XSLT into GraphViz [5] visualizer as an up-side-down tree (Figure 2b).

4.2 Loop dependence visualization

As long as the iteration space dependence graph is generated, one can visualize it in the ISV [17]. The tool shows the maximum parallelism in the data-flow execution of the loop and finds a suitable transformation interactively. First a nested loop is instrumented with output statement by `trace.xsl` to generate the symbolic memory trace. When executing the instrumented program, a memory instruction trace is generated. For instance, the instrumented Gauss-Jordan outputs the following trace:

```
<trace>
 <iteration id="0"><i>1</i><i>2</i></iteration>
 <access id="0" type="R"><var>a</var>
    <i>2</i><i>1</i></access>
```



```
N_PROGRAM
  N_COMMENT gauss-jordan
  N_PARAM
      N_PARA
        N_VAR n
        N_CONST
            const_int      16
  N_TYPE
      T. DIMENSION
      N_ARRAY_DECL (2)
          Text  N_BOUND_EXPR
        1 a     N_BOUND_EXPR (2)
        2 x     N_BOUND_EXPR (1)
  N_STATEMENT (2)
        S_DO
      1 S_DO
          N_VAR i
          N_CONST
          N_VAR n
          N_STATEMENT
              S_DO
                  N_VAR      i
                  N_CONST
                  N_VAR      n
                  N_STATEMENT
                      S_IF
                          N_EXP
                            Text   .ne.
                            N_VAR (2)
                          N_STATEMENT (2)
                            S_ASSIGN  S_DO
                          . S_ASSIGN
                          .           S_DO
      2 S_DO
          N_VAR i
```

program
comment param type statement statement end
para array_decl array_decl do do
var const bound_expr bound_expr bound_expr var const var statement var const var statement
int var exp var int do int assign
var const var const var statement var exp
int int if var var var
exp statement statement var exp var var
var var assign do var const
var exp var exp exp statement int
var var var const var const assign
var var var var int int var exp
var var var exp
var var var var
var var

Figure 2: Visualizing the AST.

```
 <access id="0" type="R"><var>a</var>
    <i>1</i><i>1</i></access>
 <access id="0" type="W"><var>f</var></access>
 <iteration id="1"><i>1</i><i>2</i><i>2</i></iteration>
 <access id="1" type="R"><var>a</var>
    <i>2</i><i>2</i></access>
 <access id="1" type="R"><var>f</var></access>
 <access id="1" type="R"><var>a</var>
    <i>1</i><i>2</i></access>
 <access id="1" type="W"><var>a</var>
    <i>2</i><i>2</i></access> ...
</trace>
```

Here only iterations with non-empty accesses are output. An access without index is a scalar, e.g., `f`.

Then the iteration space visualizer analyzes the dependences using the trace and creates the following iteration space dependence graph:

```
<ISDG>
<iteration id="1">1 2 2/iteration>
<iteration id="2">1 2 3</iteration>
<iteration id="3">1 2 4</iteration>
... (30 iteration nodes)
<dep from="1" to="13" type="flow" />
<dep from="1" to="16" type="flow" />
<dep from="1" to="19" type="flow" />
<dep from="2" to="13" type="flow" />
<dep from="2" to="13" type="anti" />
<dep from="2" to="16" type="flow" />
<dep from="2" to="19" type="flow" />
<dep from="2" to="24" type="flow" />
... (83 dependence edges)
</ISDG>
```

The ISDG is displayed in the three dimensional iteration space in ISV, as shown in Figure 3. From the graph, one sees no dependences in each i plane, i.e. the J, K loops are parallelizable.

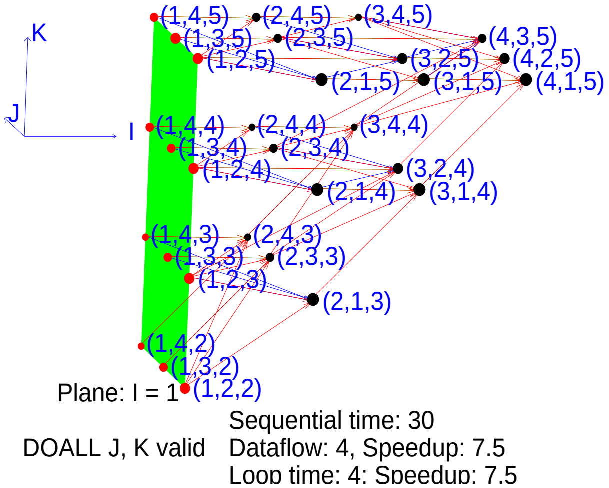


Figure 3: Gauss-Jordan visualized by the ISV in [17].

4.3 Cache miss visualization

Similar to the trace instrumentation done in the ISV, the memory trace is used to simulate a cache to generate the information for visualizing the distribution of cache misses of an program execution [16]. In ”memory trace.dtd”, instead of showing the variable and their subscripts of each access, its actual memory address is output.

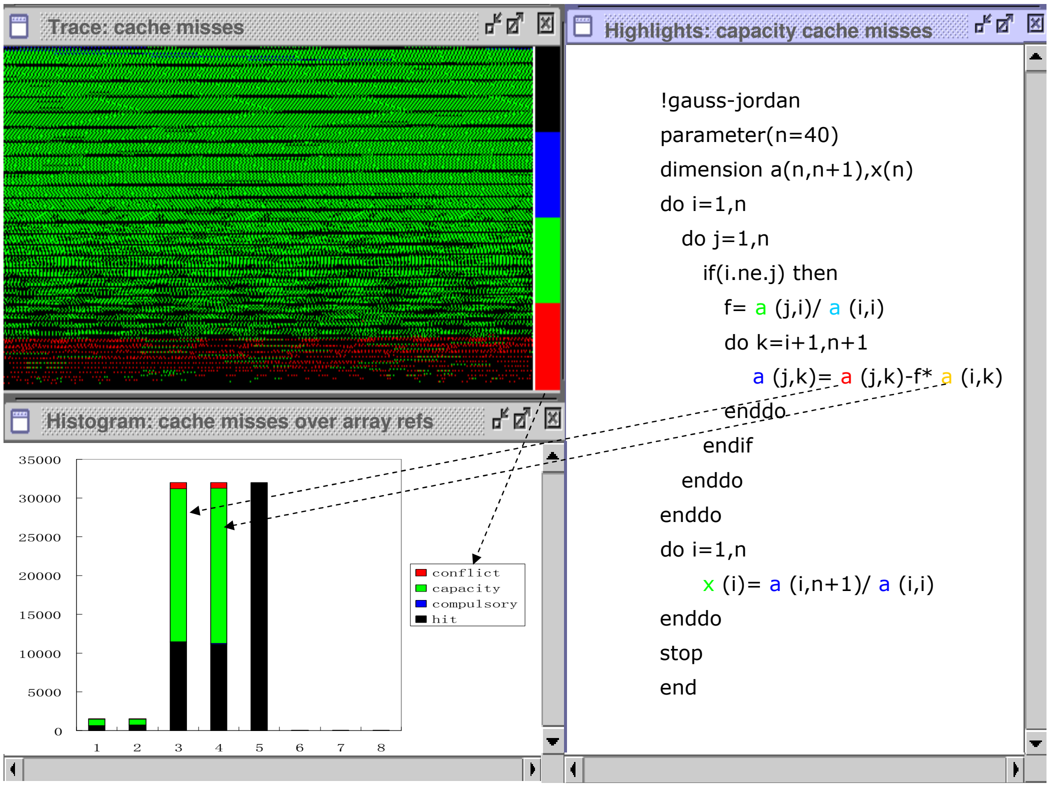
```
<!ELEMENT trace (( access )* )>
<!ELEMENT access #PCDATA ) >
<!ATTLIST access addr CDATA #REQUIRED,
                 type (R | W)  #REQUIRED>
```

Besides, the cache parameters are given to the cache simulator as an configuration file. E.g. a 1KB direct-mapped cache with cache line size 32 bytes is given.

```
<cache>
    <capacity unit="byte">1024</capacity>
    <line_size unit="byte">32</line_size>
    <associativity>1</associativity>
</cache>
```

The visualizer computes the histogram from the cache miss trace in the following format.

```
<!ELEMENT histogram (type*, item*)>
<!ELEMENT type EMPTY>
<!ATTLIST type name CDATA #REQUIRED,
               color CDATA #REQUIRED>
<!ELEMENT item (value*)>
<!ATTLIST item name CDATA #REQUIRED>
<!ELEMENT value (#PCDATA)>
```



```
!gauss-jordan
parameter(n=40)
dimension a(n,n+1),x(n)
do i=1,n
  do j=1,n
    if(i.ne.j) then
      f= a (j,i)/ a (i,i)
      do k=i+1,n+1
        a (j,k)= a (j,k)-f* a (i,k)
      enddo
    endif
  enddo
enddo
do i=1,n
    x (i)= a (i,n+1)/ a (i,i)
enddo
stop
end
```

Figure 4: Gauss-Jordan in the cache visualizer [16]. Upper-left window shows a trace of cache misses wrapped at the horizontally border; lower-left window shows a histogram chart of cache misses over array references, its legend also explains the colors used (see upper), its X-axis are array references numbered by the lexicographical order in code (see right); the array references in the right window are spectrumly colored (blue to red stands for small to large) by the intensity of capacity misses.

Both traces and histograms of the Gauss-Jordan program are displayed in figure 4. The graphs show the dominant cache misses are capacity miss, and most capacity misses occur to the 3rd and 4th references in the innermost loop.

4.4 Cache reuse distance visualization

Besides the traces and histograms, the cache visualizer also shows the distribution of the reuse distances [2] among the hot spots in the source code. When consecutive reuses of data occur far appart in time, there is a high probability for a cache miss.

First the hot spots are defined by the following DTD.

```
<!ELEMENT hotspots (source, highlight*)>
<!ELEMENT source (#PCDATA) >
<!ELEMENT hotspot (position, position)>
<!ATTLIST highlight id CDATA #REQUIRED,
                    color CDATA #REQUIRED>
<!ELEMENT position EMPTY>
<!ATTLIST position row CDATA #REQUIRED,
                   col CDATA #REQUIRED>
```

A hot spots document indicates the source file name by `source` child. Each hot spot is a region from (row_1, col_1) to (row_2, col_2) in the source code. Thus the source code is visualized with the hot spots highlighted in specified colors. The distribution of reuse distance in the source code is visualized by colors in the right window in figure 4.

Another way of presenting the hot spots is using a graph layout tool `vcg` [12]. First the cache simulator outputs the document defined by the following DTD:

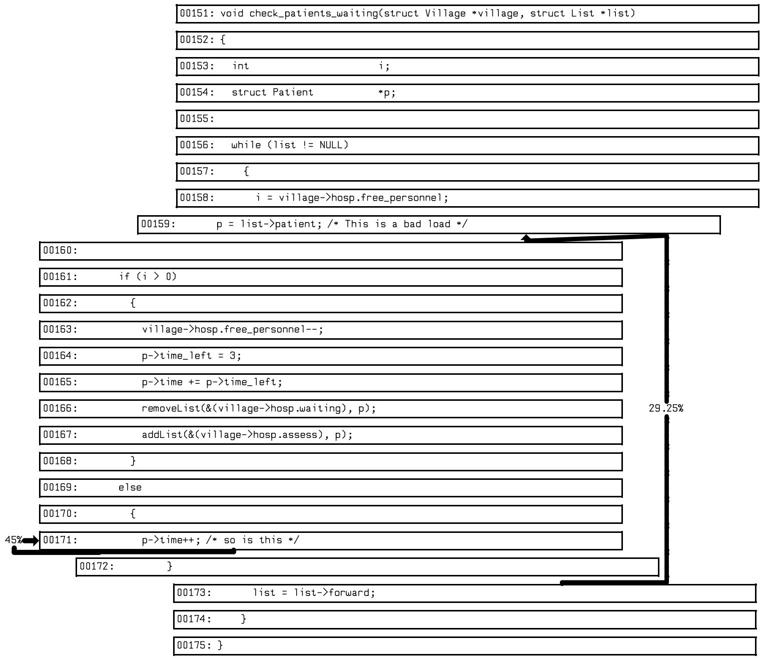
```
<!ELEMENT count (#PCDATA)>
<!ELEMENT fromid (#PCDATA)>
<!ELEMENT frompu (#PCDATA)>
<!ELEMENT function (reference*)>
<!ATTLIST function pu CDATA #REQUIRED>
<!ELEMENT log2distance (#PCDATA)>
<!ELEMENT reference (reuse*)>
<!ATTLIST reference id CDATA #REQUIRED>
<!ELEMENT reuse (log2distance, frompu, fromid, count)>
<!ELEMENT reuse_distance_graph (function+)>
```

Each instruction has a histogram counting the memory accesses over different log2 reuse distances. The reused instructions with long reuse distances that generate capacity misses are filtered, then these instructions pairs are displayed as directed graph in `vcg`.

The long reuse distances of `health`, one of the Olden benchmark programs, are visualized in figure 5. One can see these hot spots coincide with the performance observation comments in the program.

5 Conclusion

Compiler transformations use intermediate representations which were internal to the compiler. The intermediate representations such as abstract syntax tree, dependence distances, transforming matrices, loop dependence graphs, memory traces and cache reuse distances can now be exchanged in XML, because XML is extensible to ex-



```
void check_patients_waiting(struct Village *village, struct List *list)
{
  int                    i;
  struct Patient         *p;

  while (list != NULL)
    {
      i = village->hosp.free_personnel;
      p = list->patient; /* This is a bad load */

      if (i > 0)
        {
          village->hosp.free_personnel--;
          p->time_left = 3;
          p->time += p->time_left;
          removeList(&(village->hosp.waiting), p);
          addList(&(village->hosp.assess), p);
        }
      else
        {
          p->time++; /* so is this */
        }
      list = list->forward;
    }
}
```

Figure 5: The long reuse distances visualized as hot spots

press any structural data in DTD or Schema. An IR of existing compiler can be re-engineered incrementally through adding orthogonal namespaces to support additional compiler transformations. Transformations can be done even outside the compiler to exchange information with various XML-enabled programming tools. Thus, XML becomes a glue to form an open research environment by seamlessly binding the compiler with external transformation tools such as Omega calculator and visualizers for loop dependences and cache reuse distances.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison, 1986. ISBN 0-201-10088-6.

[2] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In T. Gonzalez, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, California, USA, 8 2001. IASTED.

[3] T. Bray, J. Paoli, and C. Sperberg-MacQueen. Extensible markup language. Technical report, http://www.w3.org/TR/REC-xml, feb 1998.

[4] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.

[5] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE TRANS. SOFTW. ENG.*, 19(3):214–230, May 1993.

[6] M. Girkar and C. D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[7] B. Goldberg, H. Kim, V. Kathail, and J. Gyllenhaal. The Trimaran compiler infrastructure for instruction level parallelism research. Technical report, Hewlett-Packard Laboratories, University of Illinois, NYU, 1998.

[8] D. R. Hanson. Early experience with ASDL in lcc. *Software - Practice and Experience*, 29(5):417–435, 1999.

[9] Kitware. The GCC-XML extension. Technical report, http://public.kitware.com/GCC_XML.

[10] S. Microsystems. Java object serialization specification, revision 0.9. Technical report, Sun Microsystems, 1996.

[11] W. Pugh. A practical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, AUG 1992.

[12] G. Sander. Graph layout through the VCG tool. *Lecture Notes in Computer Science*, 894:194–205, 1995.

[13] M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[14] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The zephyr abstract syntax description language. Technical Report TR-554-97, http://www.zephyr.com, Oct. 1997.

[15] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), dec 1996.

[16] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of the cache on program execution. In *Proceedings of Fifth International Conference on Information Visualization*, pages 336–341, London, England, July 2001.

[17] Y. Yu and E. D'Hollander. Loop parallelization using the 3D Iteration Space Visualizer. *Journal of Visual Languages and Computing*, 12(2):163–181, April 2001.