Proceedings of the Third IASTED International Conference
Communications, Internet, and Information Technology
November 22-24, 2004, St. Thomas, US Virgin Islands

Algorithm Partitioning and Optimization for Network Processors

Ralf Lehmann
Institute for System Architecture, Chair for Computer Networks
Technische Universität Dresden
01062 Dresden, Germany
email: lehmann@rn.inf.tu-dresden.de
Alexander Schill
Institute for System Architecture, Chair for Computer Networks
Technische Universität Dresden
01062 Dresden, Germany
email: schill@rn.inf.tu-dresden.de

ABSTRACT
Current high speed networks cannot be fully utilized by today's high end systems. The processing requirements of next generation network protocols require intelligent network cards with network protocol offload engines, for example based on a network processor. To reduce development time, we reuse existing software protocol stack implementations for partitioning and implementing on the network card. Though, manual partitioning is very time-consuming due to the complex protocol stacks. We outline an approach for tool supported software partitioning even for complex C source code.

KEY WORDS
Communication Systems, Communications Protocol, Algorithm Partitioning, Dynamic Algorithm Analysis, C Code Analysis, Network Processor.

1 Introduction and motivation

In the past few years the available network bandwidth in local area networks has grown much faster than the computing power of the connected high end computers. Thus even such high end PCs or servers cannot fully utilize local high speed networks [6]. An approach to solve this problem is partitioning the data path of the network stack and implement it on an intelligent network card based on an Intel IXP [5] network processor [1, 7].

Even if such a network processor needs special programming techniques, the reuse of existing and proven network stacks is much more efficient than the development of a completely new design for the very reason that a modern protocol stack is very complex. On the other hand due to its complexity partitioning of the data path and synchronization of signalling data between host system and network card are difficult too. Due to different memory types and buses on a IXP based network card we need support for the decision, which memory to use for each of the data structures.

In this paper we present an approach for a tool supported partitioning of the data path of a network stack and a collector for synchronization data. The main emphasis is to drastically reduce development periods connected with an improvement of network throughput and latency due to the usage of intelligent network cards based on network processors.

In the following chapter we describe a generic approach for a tool supported C source code partitioning as well as collection and rating of synchronization data even for complex implementations. Next we evaluate this approach with a prototype implementation of a set of partitioning support tools. Since a fully automatic partitioning and memory type decision is not reasonable, we discuss in the following chapter visualization possibilities of the partitioning results and it's usage for partitioning complex implementations. After classifying our approach in comparison to related work, we finally conclude our results and address future work.

2 Partitioning and optimization approach

First step for data path partitioning – independent whether either manual or automatic – should be the identification of the CPU consuming data path. When analyzing a relatively small amount of source code, manual procedure is practicable. For more complex source code like the Linux kernel [9] a more automatic way like profiling methods based on Oprofile [11], DProbes [4] or the Linux Trace Toolkit [10] should be used [6, 7].

Based on the profiling results all depending functions and subfunctions of the data path have to be partitioned. Especially if the implementation uses a wide range of dynamic functions and variables, a simple source code analysis for partitioning is impossible. The network stack of the Linux kernel for example uses these techniques very frequently.

The main idea to circumvent these problems is to use an automatic debugger, which executes the program step by step and analyzes the output of the debugger as well

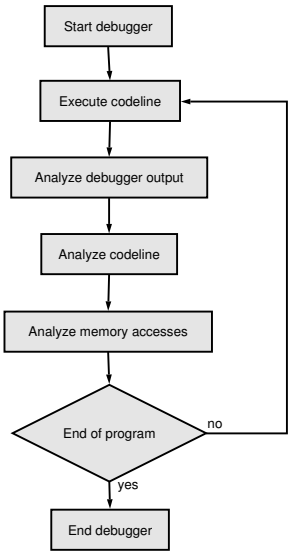

Figure 1. Analysis approach

as interpretes the active source code line in view of memory accesses. Figure 1 illustrates this approach. The main advantage of this procedure are the complete knowledge of data structures, their access, frequency of access and needed functions respectively subfunctions.

In addition to the partitioning results due to the possibilty of creating a time - memory-access - graph, statements concerning independent algorithm parts without memory read/write collisions can be made. These results can be used for dividing the algorithm onto the up to 16 microengines[1] of the IXP network processor.

On the other hand the input data has to be selected very carefully because of the heuristic charateristics of this method. Thus certain conditional forks could be not be partitioned even though they should be. The only possibility to avoid this, is to run the debugging session several times with all possible variations of input data.

3 Implementation of partitioning approach

3.1 General implementation

For the implementation of the approach the GNU debugger GDB [3] was chosen due to its flexible and powerful command line. For controlling the debugger and interpreting the debugging output a Perl [12] script was developed. The following Perl modules were used for the controlling script:

Devel::GDB – as a frontend for GDB for controlling the debugging process

[1] Microengines are parts of the processor with a special instruction set for network packet processing, which are able to run code independently

Parse::RecDescent – a module for creating a recursive grammar for C source interpretation

Parse::RecDescent::Consumer – a helper module for the RecDescent module

3.2 Automatic debugging

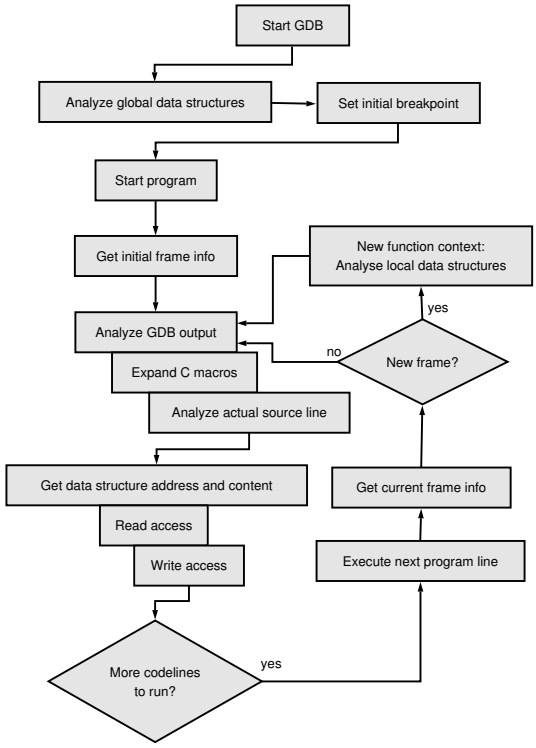

Figure 2. Flowchart of automatic debugger

The main flow of the final automatic debugging process is illustrated in figure 2. At first the debugger is started and all global data structures are recognized and logged. Since a step-by-step execution of the code is only possible if the program to be analyzed is running, next an initial breakpoint – if not defined otherwise, it's the *main* function – is set and the program started. After that the initial frame information is captured to detect function calls and its local data structures.

Thereafter the next source code line to be executed is analyzed. At first the source code line and its used C macros have to be expanded. However this feature is only available, when compiling the source code with '*-gdwarf-2 -g3*' to ensure the compiler includes preprocessor information in the debugging information.

After macro expansion the source code line is analyzed by a recursive grammar to extract all read and write accesses to data structures. Now the memory address, the content and the type of the access to the data structures are logged for later interpretation – the content of the changed variables is logged after execution of the code line.

If the programm is not finished yet or the final line to be analyzed is not reached yet, the next code line is executed and the current frame information is captured to detect a function call and therefore changed local data structures, which also have to be logged.

Now, the debugger output of the last executed step is analyzed and so on.

If there are several instructions on a source code line or one complex instruction is splitted onto several lines, the output of the GDB is insufficiently exact to achieve detailed analysis. Hence the source code is reordered by a preprocessor before compiling to ensure there is exactly on instruction per source code line.

3.3 C parsing grammar

As opposed to a common C parsing grammar, the needed grammar must be able to interpret even fragments of correct C code and to extract all read and modified data structures with one or more of the following characteristics:

- plain variables
- de-referenced pointers
- structures
- arrays and array indexes

On the other hand the used grammar does not need to check if the syntax of the code line is correct – we just trust in the C compiler. The implemented recursive grammar uses the Perl module *RecDescent*, a top-down recursive-descent text parser.

Main parts of the grammar are illustrated in figure 3.

Instruction An instruction line can consist of C commands, assign instructions or function calls, separated by one of the characters ";", "{" or "}".

C Command For correct code interpretation only a subset of all C commands has to be handled by an extra rule: comparisons, while loops and the switch construct.

Assign Instruction The rule for the assign instruction is for detection of write accesses to data structures. Every data structure left from an assign operator defined by the *assign* rule respectively every incremented or decremented data structure is marked with *write* access.

Expression The most complex rule of the grammar is the definition of an expression. At first it has to handle negates and type casts. Next case to be handled are conditional expressions. After that there is a check if a variable or a constant resp. a function call is connected via an operator with an assign instruction or an expression. Thereafter the expression is checked for

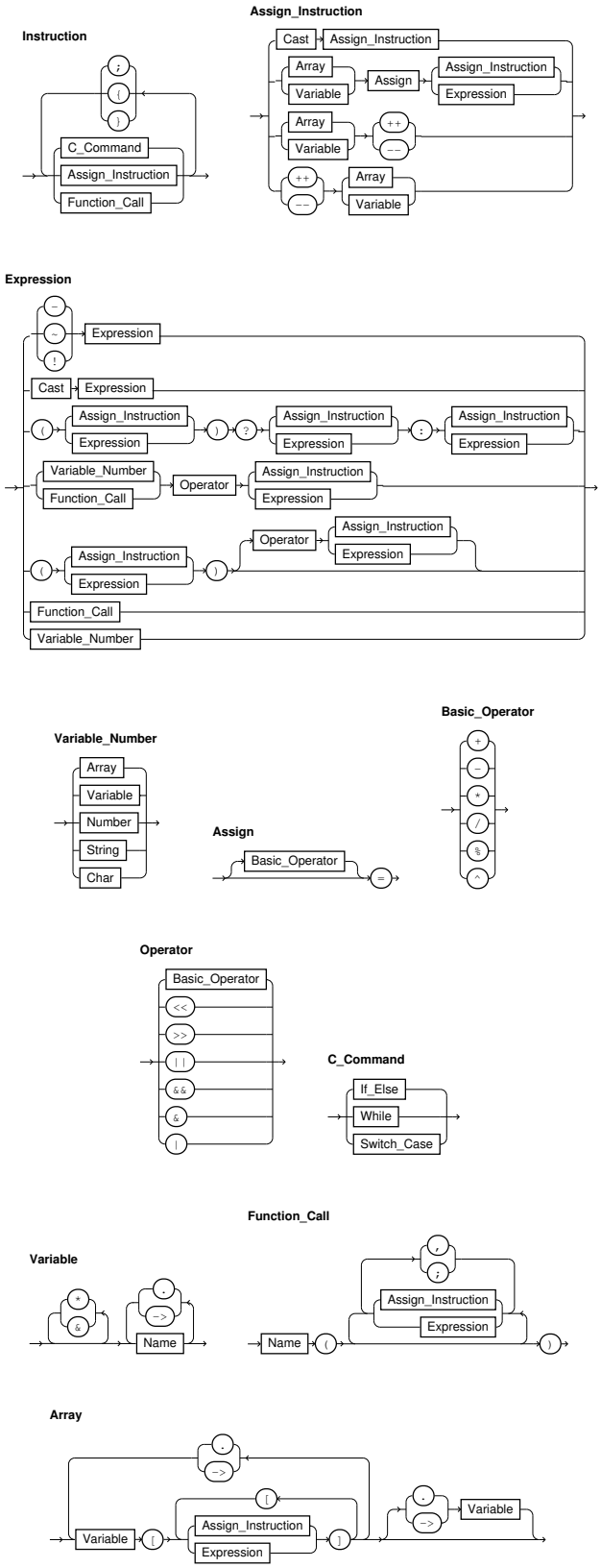

Figure 3. Cut-out of recursive grammar

```c
#define PI 3.14159265

int main() {
  float radius = 10.0;
  float height = 5.0;
  float volume, diameter, area, circumfence;

  diameter =  2.0 * radius;
  area = radius * radius * PI;
  circumfence = diameter * PI;
  volume = area * height;

  printf("Cylinder\n");
  printf("Diameter:    %f\n", diameter);
  printf("Footpoint:   %f\n", area);
  printf("Circumfence: %f\n", circumfence);
  printf("Volume:      %f\n", volume);

  volume = (PI / 3.0) * radius * radius * radius * height;

  printf("\nCone\n");
  printf("Volume:      %f\n", volume);

  return 0;
}
```

Figure 4. Demonstration program

a parenthesis construct with or without another operation. Finally an expression may consist of a plain function call or a variable resp. constant.

Every data structure found in an expression is marked with *read* access.

Variable Number This rule defines all possibilities for data structures or constants, arrays, variables, numbers, strings and characters.

Assign The assign rule is for definition of a plain assignment or an assignment together with an basic operation.

Basic Operator A basic operator is one of the operations “+”, “-”, “*”, “/”, “%”, “^” that may be connected with an assignment.

Operator The rule for an operator checks for a basic operator and bit or logical operations.

Function Call A function call may consist of a function name followed by a left parenthesis, an optional list of parameters and a right parenthesis.

Variable A variable name is either a plain name or a list of names connected with a “.” or a “->” depending of the type of the structure. A variable name can also start with referencing (“*”) or de-referencing (“&”) signs.

Array Finally an array construct may consist of a variable and one or more indexes – an assign instruction or an expression – each embedded in brackets or in case of a structure a list of such basic arrays connected with a “.” or a “->” and a possible variable ending.

3.4 Debugger output

The run of the automatic debugger results in a XML output with detailed information about every executed code line:

```
0:
   create: 0x8048564 = (const int)131073 (_IO_stdin_used)
1:        float radius = 10.0;
   create: 0xbffff8c4 = (float)10 (radius)
   write:  0xbffff8c4 = (float)10 (radius)
2:        float height = 5.0;
   create: 0xbffff8c0 = (float)5 (height)
   write:  0xbffff8c0 = (float)5 (height)
3:        diameter =  2.0 * radius;
   create: 0xbffff8b8 = (float)20 (diameter)
   read:   0xbffff8c4 = (float)10 (radius)
   write:  0xbffff8b8 = (float)20 (diameter)
4:        area = radius * radius * 3.14159265;
   create: 0xbffff8b4 = (float)314.159271 (area)
   read:   0xbffff8c4 = (float)10 (radius)
   write:  0xbffff8b4 = (float)314.159271 (area)
5:        circumfence = diameter * 3.14159265;
   create: 0xbffff8b0 = (float)62.831852 (circumfence)
   read:   0xbffff8b8 = (float)20 (diameter)
   write:  0xbffff8b0 = (float)62.831852 (circumfence)
6:        volume = area * height;
   create: 0xbffff8bc = (float)1570.79639 (volume)
   read:   0xbffff8b4 = (float)314.159271 (area)
   read:   0xbffff8c0 = (float)5 (height)
   write:  0xbffff8bc = (float)1570.79639 (volume)
7:        printf("Cylinder\n");
8:        printf("Diameter:    %f\n", diameter);
   read:   0xbffff8b8 = (float)20 (diameter)
9:        printf("Footpoint:   %f\n", area);
   read:   0xbffff8b4 = (float)314.159271 (area)
10:       printf("Circumfence: %f\n", circumfence);
   read:   0xbffff8b0 = (float)62.831852 (circumfence)
11:       printf("Volume:      %f\n", volume);
   read:   0xbffff8bc = (float)1570.79639 (volume)
12:       volume = (3.14159265 / 3.0) * radius * radius * radius * height;
   read:   0xbffff8c4 = (float)10 (radius)
   read:   0xbffff8c0 = (float)5 (height)
   write:  0xbffff8bc = (float)5235.98779 (volume)
13:       printf("\nCone\n");
14:       printf("Volume:      %f\n", volume);
   read:   0xbffff8bc = (float)5235.98779 (volume)
15:       return 0;
16:    }
```

Figure 5. Output of automatic debugger

- source code line with expanded macros and file information
- created data structures with information about type, name, memory address and if availabe, the content
- read data structures with information about name, memory address and content
- written data structures with information about name, memory address and new content
- in case of a function call, function name, parameters as created data structures

On the basis of this XML output, all executed code lines needed for this algorithm are logged. Additionally a summary of needed data structures and the frequency of accesses is available.

Figure 4 shows a simple C program for demonstration of the automatic debugger. It calculates some simple values for a cylinder and a cone.

The analysis output of the debugging process is shown in figure 5. For every executed code line – marked with a counter – the created, read and written variables are quoted including memory address, type, name and value. Since the name of the data structure is not expressive enough, all logging is indexed by the memory address of the data. So even C pointer magic with referenced and dereferenced variables and access to one specific memory address by different variables with different names can be detected and logged.

```
Address:    bffff8bc
    create: 6
    type:   float
          read:  11 :: volume = 1570.79639
          read:  14 :: volume = 5235.98779
          write:  6 :: volume = 1570.79639
          write: 12 :: volume = 5235.98779
Address:    bffff8c0
    create: 2
    type:   float
          read:   6 :: height = 5
          read:  12 :: height = 5
          write:  2 :: height = 5
Address:    bffff8b0
    create: 5
    type:   float
          read:  10 :: circumfence = 62.831852
          write:  5 :: circumfence = 62.831852
Address:    bffff8c4
    create: 1
    type:   float
          read:   3 :: radius = 10
          read:   4 :: radius = 10
          read:  12 :: radius = 10
          write:  1 :: radius = 10
Address:    bffff8b4
    create: 4
    type:   float
          read:   6 :: area = 314.159271
          read:   9 :: area = 314.159271
          write:  4 :: area = 314.159271
Address:    bffff8b8
    create: 3
    type:   float
          read:   5 :: diameter = 20
          read:   8 :: diameter = 20
          write:  3 :: diameter = 20
Address:    8048564
    create: 0
    type:   const int
```

Figure 6. Summary of memory accesses

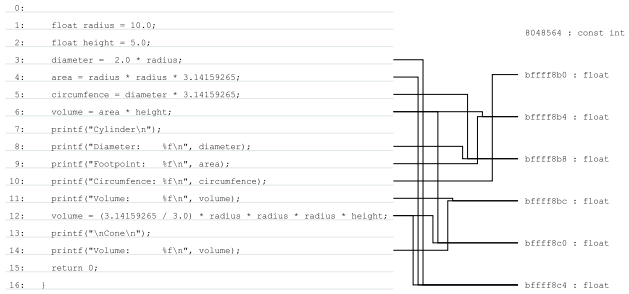

Figure 7. Algorithm and memory read access graph

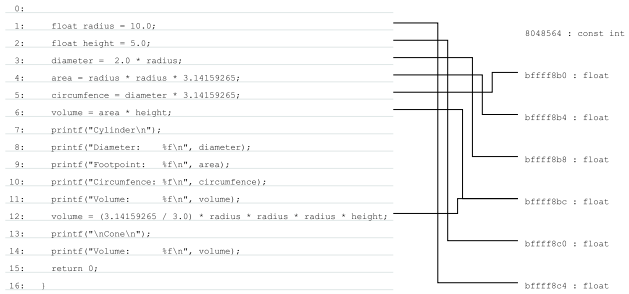

Figure 8. Algorithm and memory write access graph

The summary of memory accesses is shown in figure 6. For every memory address it lists the data type, code line of first usage (creation) and all the read respectively write accesses with the corresponding code lines.

On the basis of this summary, all data needed for synchronization between a parent and the separated algorithm, that is implemented on the network processor, is available.

Even such a simple and short programm generates a relatively complex output; if there are more instructions with more data structures, the output of the analysis must be visualized in another way.

4 Visualization and usage of analysis results

4.1 Visualization of analysis results

Based on the XML output of the debugger, detailed graphs about program or algorithm function and memory accesses can be created.

Figures 7 and 8 show an algorithm and memory read respectively write access graph for the program shown in figure 4. Every executed code line – placed on the left – has a connection to every accessed memory address – placed on the right –.

Even if the program is really short and clear, the graph to illustrate the access to data structures gets rapidly complex especially when combining both graphs. At the end the graphs are too complicated to get results for partitioning and especially for parallelizing.

Other problems are the usage of conditional forks and dynamic dependences of input data in the algorithm implementation. To achieve an optimal prognosis for partitioning and to get all relevant code lines in that case, the automatic debugging has to be run multiple times with all possible input data. Now all results can be compared – the frequency of the different executed code lines reflects the most important lines to be implemented on the network processor. To find the best moment for resynchronization with the host implementation the debugging system has to run with special input data to force a situation for a planned fallback from the network processor implementation to the host system.

4.2 Usage of analysis results

The main application for this approach is to get partitioning results for highly complex implementations, for instance a network protocol stack implementation with the aim of an optimal as possible implementation on a network processor.

Therefore a more interactive analysis is needed since the visualization as done in figures 7 and 8 gets confusing for an analysis for instance of a protocol stack. One possibility is to use a special graph browser with the following functionality:

- If code lines are tagged manually, every conflicting line with variable write access to one of the accessed data structures, used in the marked area, is highlighted. On that basis it is possible to find independent

code parts and the latest moment of synchronization of parallel running parts of the algorithm.

- On the basis of identified conflicts concerning memory write accesses, suggestions for possible partitioning onto the microengines are made.

Final aim of this procedure is to divide the code onto all available microengines of the network processor and define all synchronization points and data.

A fully automatic parallelizing is possible as well, but should only be an assistance for the interactive browsing due to special needs of the microengines of the network processor.

5 Related work

An approach for software/ hardware partitioning of C code is described in [8, 2]. Their aim is to accelerate the execution of code using dynamically reconfigurable processors. The original Nimble compiler [8] focuses on partitioning of only loops and their optimization. However the Garp C compiler [2] tries to partition the code into basic blocks without branches into or out of the middle. For an acceleration of a network stack using network processors this approach is too fine grained with no predication on frequency of memory accesses.

A data structure analysis of a C implementation is presented in [13]. However, they only capture and explore single program states but offer a three dimensional visualization of all used data structures of a program at one specific state. For a partitioning approach for network processors the frequency and moment of memory accesses is missing.

Due to their object oriented focus approaches for aspect oriented programming and object oriented approaches for visualization and development of programs are not transferable to dynamic C code analysis.

6 Conclusions and future work

For easing the burden of network protocol processing on today's host processors partitioning approaches with usage of protocol offload engines are useful. However partitioning an existing software implementation of a protocol stack is too complex for manual procedure. Using an automatic debugger for data collection is a possible approach for algorithm and C code analysis. These data about code execution and moment and frequency of memory access are the basis for optimal algorithm partitioning on microengines of a network processor.

Since the automatic debugger and especially the C grammar is written in Perl, timeouts in protocol processing may occur on code execution and analysis. Therefore the debugger has to be optimized and accelerated itself. Future work also includes the analysis and complete partitioning of the Linux TCP/IP network stack for the IXP network processor.

References

[1] Mirko Benz and Ralf Lehmann. TCP Acceleration based on Network Processors. In *IASTED International Conference on Communications, Internet, and Information Technology (CIIT)*, St. Thomas, US Virgin Islands, 2002.

[2] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33:4:62–69, April 2000.

[3] *GDB: The GNU Project Debugger – http://www.gnu.org/software/gdb*. Internet WWW document.

[4] IBM Linux Technology Center. *Dynamic Probes for Linux – http://oss.software.ibm.com/-developer/opensource/linux/projects/dprobes/*, 2002. Internet WWW document.

[5] Intel Corp. *Intel Network Processors – http://www.intel.com/design/network/products/-npfamily/*. Internet WWW document.

[6] Ralf Lehmann and Mirko Benz. Analysis of TCP/IP Protocol Processing in Gigabit Networks. In *Proceedings of the 2002 WSEAS International Conference on Information Security, Hardware/Software Codesign, E-Commerce and Computer Networks*, Rio de Janeiro, Brazil, October 2002.

[7] Ralf Lehmann, Mirko Benz, Stephan Groß, and Maik Hampel. IPsec Protocol Acceleration using Network Processors. In *IASTED International Conference on Communications, Internet, and Information Technology (CIIT)*, Scottsdale, Arizona, November 2003.

[8] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Proceedings of the 37th Design Automation Conference*, pages 507–512, Los Angeles, CA, 2000.

[9] *The Linux Kernel Archives – http://www.kernel.org*. Internet WWW document.

[10] Opersys. *The Linux Trace Toolkit – http://www.opersys.com/LTT/*, 2002. Internet WWW document.

[11] *OProfile – http://oprofile.sourceforge.net*. Internet WWW document.

[12] *The Perl Directory – http://www.perl.org*. Internet WWW document.

[13] T. Zimmermann and A. Zeller. Visualizing Memory Graphs. In *Proceedings of the Dagstuhl Seminar 01211 "Software Visualization"*, Dagstuhl, Germany, May 2001.