Monitoring the Structure and Behavior of Programs

Edil S. T. Fernandes and Valmir C. Barbosa

Programa de Engenharia de Sistemas e Computação
COPPE—Federal University of Rio de Janeiro—UFRJ
edil, valmir@cos.ufrj.br

Abstract

The paper presents some experiments involving the Basic Block Machine Model (BBMM for short). Instead of a single instruction, the objects processed by this execution model are those groups of machine instructions which define a basic block.

As a consequence of the detailed simulation level required by the model, many characteristics of the application programs were obtained. For example, during the simulation of the integer programs of the SPEC95 suite, we found that more than 50% of the instructions of application programs remain untouched along the whole execution. In addition to that, the experiments showed that less than 2% of the basic blocks are responsible for more than 90% of the instruction count.

1. Introduction

Compilers, Operating Systems, Libraries and other system programs can be characterized by their generality: they must be prepared to handle the occurrence of a large class of events, including those which are unlikely to occur.

For example, instead of crashing the whole computing system, an Operating System must detect an unexpected input, and discontinue the corresponding application program. At execution time, the portions of the object code dedicated to these events will seldom be executed. However, for the sake of the system’s safety, these portions must be within the object code.

The actual code implementing the user’s algorithm presents the same behavior: the vast majority of the instructions will never be executed. Since only a small fraction of the program is responsible for the whole execution time, then we can understand why the following architectural techniques are so efficient: the instruction caches present very high hit ratios; the trace cache mechanism can be very efficient; an alternative program layout can execute faster, and so on.

For example, the Compress program (of the SPEC95 benchmark) is formed by 12,979 instructions. Our experiments with BBMM revealed that 72.76% of its instructions remained untouched during the whole execution (35,818,718 instructions executed). In other words, only 3,535 instructions were responsible for the instruction count.

A group of instructions is the standard unit of processing in BBMM. This group forms a basic block that is defined as:

Definition. A basic block is an ordered collection of machine instructions with no entry points, except the first instruction of the block, and no branches, except possibly the last instruction of the block [1].

The paper is organized into six sections. The next section shows the difference between the conventional execution model and the basic block execution model. Section 3 describes how an object program can be transformed into a two-dimensional structure. Section 4 explains how our machine was simulated. The section also shows the results produced by the experiments. Section 5 presents related work and summarizes the main contributions of the paper. Section 6 contains conclusions.

2. Motivation

The BBMM concept can be applied to any ISA (Instruction Set Architecture), with two stages being required:

• code generation as a two-dimensional structure;
• execution of an entire basic block per cycle.

Existing compilers and assemblers generate object code in the traditional fashion, i.e., the individual instructions are the standard execution units.

Since BBMM considers an entire basic block as its standard unit, it is necessary to pre-process existing executable programs before the actual execution. During this stage, the basic blocks of the programs are iden-
tified and separated, and a file containing the basic blocks is generated.

This new executable file contains all the instructions of the original one, but instead of including only one instruction per line it contains a whole basic block per line. Figure 1 illustrates the organization of a BBMM executable program.

![Figure 1: An Object Program as a Two-dimensional Structure](image)

Figure 1 shows the first seven basic blocks of a hypothetical program. Each small rectangle represents one instruction of the basic block \( b_i \). The next section explains how a conventional object program is converted into the two-dimensional structure required by our Basic Block Machine.

It is always possible to obtain the two-dimensional structure of Figure 1 from the executable code of any ISA.

3. Basic Block Identification

In order to construct the two-dimensional structure required by our model, we need to identify the basic blocks within the executable program.

A very large fraction of the basic blocks of the program can be found in a straightforward way through the identification of control transfer instructions (conditional and unconditional branches, call and return commands, supervisor calls and returns, traps, and so on). These control transfer instructions can define up to four basic blocks:

- two blocks delimited by the control transfer—since a control transfer is the last instruction of a basic block, then the following instruction is the first instruction of the next basic block;
- and two blocks delimited by the target address of the control transfer—in this case, the target address is an entry point. Consequently, this address is the first instruction of a basic block and the previous instruction finishes the preceding block.

Some blocks are harder to find because the target address of the control transfer instruction can be within a register (transfers like: "jump and link register," and "jump register"). In our environment, these target addresses are captured at execution time by an algorithm which recognizes the two new basic blocks defined by the target address.

Another way to detect these blocks is to use static techniques, as that described by C. Cifuentes and M. V. Emmerik [2] (the technique is able to recover jump tables and their target address in a machine- and compiler-independent way).

The main data structure used by BBMM is a two-dimensional structure containing the basic blocks of the application program. For each basic block, the structure has one entry storing the instructions of the block, the addresses of the first and the last instruction of the block, the addresses of the instructions which are the ancestors of the current block, and the basic block count.

The structure generation requires two steps. In the first step, the object code is examined and the control transfer instructions detected. If possible, the initial addresses of two basic blocks are inserted into the entry. Otherwise, we have an "indirect jump" and only at execution time will the target address be determined.

4. Monitoring Basic Block Programs

We have mentioned that BBMM is a conceptual model which can be applied to any existing ISA. In this paper we will show some results related to the variation of the MIPS IV architecture which is described in the SimpleScalar Tool Set [3]. The sim-fast functional simulator together with the integer programs of the SPEC95 suite of that distribution were used in our experiments.

During each cycle of BBMM, the basic block counter is incremented and the sim-fast is invoked to process an entire basic block. At the end of the execution of each block, if the control is not transferred to the start of another basic block, that means that a new block
has been detected. In this case, the two-dimensional structure is updated.

Whenever the control is transferred to the first instruction of another basic block, we need to check if the address of the current block belongs to the ancestor list of the next block. If appropriate, the block address is included in the ancestor list. Figure 2 shows how a two-dimensional structure is processed by our execution model.

![Diagram of 2-Dimensional Structure Processing](image)

**Figure 2: Dynamic Detection of Basic Blocks**

In Figure 2, BBMM-Sim is the simulator of our model. This portion activates the sim-fast and checks the basic block boundaries. The new two-dimensional structure can be used by subsequent executions of the same program. Successive executions of the same program (with different input data) generate new two-dimensional structures which may be more updated.

During each execution, new basic blocks can be detected and included in the structure. The same can occur with the lists of ancestors because more elements can be introduced. Table I shows the number of basic blocks containing from one up to six ancestors. The last column of the table shows the maximum number of ancestors present by one basic block of the corresponding SPEC95 program.

**Table I: Ancestors of Basic Blocks**

<table>
<thead>
<tr>
<th>Prog.</th>
<th>1-anc</th>
<th>2-anc</th>
<th>3-anc</th>
<th>4-anc</th>
<th>5-anc</th>
<th>6-anc</th>
<th>...</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>2,859</td>
<td>400</td>
<td>128</td>
<td>55</td>
<td>22</td>
<td>11</td>
<td>...</td>
<td>47</td>
</tr>
<tr>
<td>Gcc</td>
<td>65,135</td>
<td>8,099</td>
<td>2,234</td>
<td>1,293</td>
<td>659</td>
<td>390</td>
<td>...</td>
<td>1,490</td>
</tr>
<tr>
<td>Go</td>
<td>11,631</td>
<td>1,866</td>
<td>514</td>
<td>229</td>
<td>127</td>
<td>59</td>
<td>...</td>
<td>378</td>
</tr>
<tr>
<td>Ijpeg</td>
<td>8,857</td>
<td>1,314</td>
<td>254</td>
<td>100</td>
<td>59</td>
<td>21</td>
<td>...</td>
<td>141</td>
</tr>
<tr>
<td>Li</td>
<td>5,113</td>
<td>642</td>
<td>193</td>
<td>86</td>
<td>29</td>
<td>25</td>
<td>...</td>
<td>91</td>
</tr>
<tr>
<td>M88ksim</td>
<td>8,312</td>
<td>1,147</td>
<td>350</td>
<td>128</td>
<td>74</td>
<td>44</td>
<td>...</td>
<td>280</td>
</tr>
<tr>
<td>Perl</td>
<td>16,315</td>
<td>1,737</td>
<td>546</td>
<td>234</td>
<td>100</td>
<td>60</td>
<td>...</td>
<td>266</td>
</tr>
<tr>
<td>Vortex</td>
<td>24,193</td>
<td>2,228</td>
<td>934</td>
<td>347</td>
<td>184</td>
<td>125</td>
<td>...</td>
<td>1,557</td>
</tr>
</tbody>
</table>

Table II lists the instruction count, the number of instructions, and the total of basic blocks of each program. The last two columns of the table were generated by successive executions of the programs (the whole "reference" input set was used to determine the number of basic blocks of each program). The "training" set was used as the input data to determine the instruction count and the code usage of the programs.

The simulator of the Basic Block Machine Model provides the instruction usage in a straightforward way: to execute a program in this model, BBMM-Sim passes to the 'sim-fast' simulator the boundaries of the next basic block; when the control is returned, BBMM-Sim increments the corresponding basic block count and a new BBMM cycle starts.

At the end of the simulation, each count in the two-
dimensional structure contains the number of times the block was executed. A basic block that remained untouched along the whole execution has its count equal to zero.

We found that the object code usage of integer programs of the SPEC95 is very modest. Our execution model detected that more than 50% of the instructions remained untouched, during the whole execution, for the vast majority of the integer programs. Figure 3 gives the percentages of instruction usage.

![Figure 3: Instruction Usage (%)]

The dark bars in Figure 3 give the percentages of the instructions which were executed at least once. They range from 17.44% (M88ksim) up to 49.13% (Vortex) for the vast majority of the programs. The Go program was the unique exception: 70.05% of its instructions were executed at least once.

Similar behavior occurred with the usage of the basic blocks: more than 50% of the basic blocks remained untouched during the whole execution. Again, the Go program was the exception, with only 35.62% of its basic blocks remaining untouched. Figure 4 shows the usage of basic blocks.

![Figure 4: Basic Block Usage (%)]

The percentages of untouched blocks range from 55.60 (Vortex) up to 85.69 (M88ksim). Examining Figures 3 and 4, one can see that the percentages of untouched instructions are greater than the corresponding percentages of blocks. The main reason for this difference is related to the dynamic average length of basic blocks.

The specialized literature indicates that the static average length of basic blocks is around four to five instructions.

On the other hand, our experiments show that at execution time, an expressive number of basic blocks with less than four instructions are the most executed ones.

Table III shows the number of basic blocks containing from one up to five instructions (columns labeled \(b_1, \ldots, b_5\)). The last column of the table indicates the number of basic blocks containing more than five instructions.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>(b_1)</th>
<th>(b_2)</th>
<th>(b_3)</th>
<th>(b_4)</th>
<th>(b_5)</th>
<th>&gt; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>162</td>
<td>211</td>
<td>168</td>
<td>110</td>
<td>66</td>
<td>174</td>
</tr>
<tr>
<td>Gcc</td>
<td>6,481</td>
<td>9,297</td>
<td>6,143</td>
<td>4,229</td>
<td>2,730</td>
<td>56,890</td>
</tr>
<tr>
<td>go</td>
<td>1,358</td>
<td>1,457</td>
<td>1,193</td>
<td>956</td>
<td>848</td>
<td>3,572</td>
</tr>
<tr>
<td>Ijpeg</td>
<td>440</td>
<td>577</td>
<td>443</td>
<td>381</td>
<td>248</td>
<td>810</td>
</tr>
<tr>
<td>Li</td>
<td>253</td>
<td>371</td>
<td>377</td>
<td>214</td>
<td>110</td>
<td>334</td>
</tr>
<tr>
<td>M88k</td>
<td>267</td>
<td>366</td>
<td>232</td>
<td>175</td>
<td>108</td>
<td>307</td>
</tr>
<tr>
<td>Perl</td>
<td>679</td>
<td>970</td>
<td>828</td>
<td>430</td>
<td>297</td>
<td>810</td>
</tr>
<tr>
<td>Vortex</td>
<td>3,032</td>
<td>3,290</td>
<td>1,082</td>
<td>775</td>
<td>660</td>
<td>3,763</td>
</tr>
</tbody>
</table>

The values presented in Table III only provide the length of the basic blocks which were executed at least once. Summing up the values in a row of this table yields the overall number of blocks which were executed at least once.

However, the contribution given by some blocks are much more significant than that of others (because they were executed many times).

For example, huge blocks, formed by thousands of instructions but executed just a few times, give a modest contribution to the instruction count. The combination of the basic block length with its execution frequency is a better metric to assess the contribution of the block to the instruction count.

In other words, the contribution of each block is a fraction of the instruction count and is equal to the product “block count \times block length.”

In order to evaluate the contribution given by blocks with different lengths, we have sorted these products in decreasing order.

Table IV presents the contribution of the basic blocks (containing from 1 up to 5 instructions) to reach
90% of the instruction count of the corresponding program.

**Table IV: Basic Block Contribution (%)**

<table>
<thead>
<tr>
<th>Prog</th>
<th>b₁</th>
<th>b₂</th>
<th>b₃</th>
<th>b₄</th>
<th>b₅</th>
<th>&gt;5 blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>20.27</td>
<td>19.47</td>
<td>9.87</td>
<td>2.39</td>
<td>7.42</td>
<td>31.00</td>
</tr>
<tr>
<td>Gcc</td>
<td>22.91</td>
<td>20.98</td>
<td>13.98</td>
<td>13.58</td>
<td>6.94</td>
<td>11.62</td>
</tr>
<tr>
<td>Go</td>
<td>19.97</td>
<td>16.40</td>
<td>12.78</td>
<td>8.07</td>
<td>9.69</td>
<td>23.11</td>
</tr>
<tr>
<td>Ijpeg</td>
<td>7.95</td>
<td>21.82</td>
<td>12.93</td>
<td>0.81</td>
<td>13.83</td>
<td>32.90</td>
</tr>
<tr>
<td>Li</td>
<td>8.55</td>
<td>14.91</td>
<td>26.54</td>
<td>5.59</td>
<td>12.76</td>
<td>21.72</td>
</tr>
<tr>
<td>M88k</td>
<td>14.00</td>
<td>21.64</td>
<td>6.90</td>
<td>3.50</td>
<td>5.44</td>
<td>38.57</td>
</tr>
<tr>
<td>Perl</td>
<td>15.48</td>
<td>13.96</td>
<td>15.66</td>
<td>4.24</td>
<td>9.82</td>
<td>30.86</td>
</tr>
<tr>
<td>Vortex</td>
<td>16.63</td>
<td>18.19</td>
<td>16.15</td>
<td>1.62</td>
<td>7.16</td>
<td>30.26</td>
</tr>
</tbody>
</table>

The contribution given by blocks with more than five instructions is listed in the column labeled with "> 5." The last column of Table IV lists the number of basic blocks responsible for 90% of the instruction count.

As one can see, just a few blocks account for that percentage. For example, Ijpeg is formed by 10,662 blocks (see Table II), and only 42 of them were responsible for 90% of the instruction count. Blocks containing one and two instructions were responsible for a large fraction of the instruction count.

5. Related Work and Contribution

Our work deals with the structure and behavior of application programs. A pioneering work in this area was developed by D. E. Knuth [5]: the author presented an empirical study of FORTRAN programs which is very useful for code optimization.

In the basic block processing area, there is the work developed by S. Melvin and Y. Patt [6]. The authors proposed a basic block processor, and were interested in improving instruction scheduling with their execution model.

In 1990, K. Pettis and R. Hansen presented a work describing a technique for code positioning, which was based on the execution profile of the program [7]. Like other code layout reorganization methods, this technique improves the performance of the processor because it reduces both the number of page faults and the misses in the instruction cache. Recently, there has been a growing interest in the “memory gap problem” [4]. For this reason, several techniques to reorganize the code layout are being investigated (see [8–10]).

We have seen in this paper that the vast majority of the instructions (and the majority of basic block as well) of a program remains untouched during the whole execution. As mentioned in [11], the low amount of useful instructions in the object program has many implications both for the optimization of programs and organization of future processors. For example, the instruction caches and fill units will be much more efficient if the usage and the boundaries of the basic blocks are considered.

This characteristic of programs explains why code positioning techniques improve the processing time: since the untouched portions of the code are kept separate from the executed ones, then most of the instructions in the cache are either being executed or were executed in a previous cycle and perhaps will be executed in the future. The occasional presence of untouched instructions in the cache is due to the fetch width of the architecture: instructions belonging to different basic blocks (including the instructions from an untouched block) can be transferred to the cache by the fetch operation.

The reorganization of the code layout is a complex task because it may involve the modification of many target addresses of control transfers. From our studies we have found that it is very useful to keep the untouched blocks apart from the others.

6. Conclusions

An execution model which treats the basic blocks as the standard unit of processing was presented and simulated. Through this model, many behavioral characteristics of application programs were investigated.

A modified version of the MIPS-IV instruction set architecture together with the SimpleScalar Tool Set [3] were adopted in the implementation of our execution model. The integer programs of the SPEC95 suite were used in our experiments, with the “training” and “reference” input collections being processed completely.

Our experiments revealed that the vast majority of the instructions of a program remains untouched during the whole execution. The basic blocks present a similar behavior: less than 50% of the blocks are the executed ones. One can argue that this behavior of application programs is already known by the experts on code optimization. From a qualitative viewpoint, that is true. However, from a quantitative perspective, there has been no previous, public paper giving the actual instruction and basic block usage of programs (except the rule of thumb stating that “10% of the code is responsible for 90% of the instruction count”).

This low instruction usage is responsible for the effectiveness of code reordering techniques. On the other hand, examining the number of ancestors of the basic blocks provided in the paper, we can conclude that a
complete reorganization of the code, based on the execution profile, is a hard task (because there are blocks with too many ancestors).

The Basic Block Machine Model employs a two-dimensional structure holding the object program and other fields to store the block count and the addresses of the ancestors of blocks. Using this structure, we have determined many dynamic characteristics of the programs.

Now we already know that blocks containing up to three instructions are responsible for a large fraction of the instruction count, and only a few basic blocks are responsible for more than 90% of the instruction count. The percentages of these blocks are less than 2% for six of the test programs. These values are very useful in sizing the trace cache structure of experimental processors.

The implementation of the BBMM concept with another instruction set architecture is feasible. Now we are studying the implementation of this concept on the x86 instruction set architecture. Our investigation continues.

References


