Working Sets at Function Level

Ravi Batchu Saul Levy

Department of Computer Science

Rutgers University

{batchu,levy}@cs.rutgers.edu

Abstract

The trend in computer architecture is that of an increasing gap between rapidly increasing processor speeds and the relatively slower memory subsystems. Program locality has played a crucial role in engineering economically viable computer systems. While program locality has been studied and exploited at different levels of memory hierarchy, including the cache, *block* and page levels, there has been little or no effort to study locality at the level of functions and to leverage it.

In this paper we show that there is a strong correlation between current function references and those in the immediate future, whereas, those in the distant future tend to be more or less uncorrelated. In particular we show that an average working set (WS) size of 20% of the program size in terms of number of functions is sufficient, in all cases, to ensure a miss rate of less than 2%. Moreover, we also observe that in most cases a WS of half that size is good enough to ensure similar low miss rates. We also compare two different kinds of WSs – fixed size maintained by LRU and variable size based on a WS parameter – and show that the variable size WS consistently performs better. Finally we present a novel way of exploiting the locality at function level.

Keywords: Program Behavior, Working Set, Temporal Locality, Function/Procedure Reordering

1 Introduction

One of the oldest and most widespread styles of building software is that of top-down programming, in which a divide and conquer strategy is used to repeatedly break down the given problem until the sub-problems resulting from such decomposition are small enough to be easily solved and implemented as individual functions. The intent of this paper is to show that there is a considerable amount of temporal locality in the usage of functions and that the references to the functions in the immediate future can be predicted from the current pattern of usage. While the page forms the level of granularity at which pieces of program are brought into physical memory, the functions are perhaps more indicative of the program's immediate needs. This observation is especially important, due to the growing bottleneck between the processor and memory. In this paper we also outline a mechanism to dynamically relocate functions to facilitate the exploitation of locality at the level of functions.

The disparity in page size and function size is unlikely to vanish. The page size is around 4K and has been determined by the following factors:

i. Transfer from the disk into physical memory is slow compared to the physical memory access and there is a fixed overhead of the seek time for every disk access. A single transfer of a large page is more desirable than multiple transfers of smaller pages from the disk.

ii. Smaller page size results in larger page tables. The page size has to be reasonably large so that the overhead of the additional space for the page tables is only a small fraction of the amount of space covered by the pages.

iii. Larger page size results in fewer TLB misses.

iv. Larger page size results in internal fragmentation. Some of the portions of the page brought in may not be needed.

Constant improvements in the semiconductor technology are resulting in faster CPUs and memory chips. However, due to the inherent mechanical nature of the disks, the disk access time is not likely to show such improvements. The influence of (i) is, thus, going to increase. The move to 64 bit processors with a larger address space would require larger page tables with the current page size. This would increase the influence

of (ii) also. Therefore, it is unlikely that the page size is going to decrease. Function size is influenced by ease of implementation and management of code and is much smaller, say a screenful of lines in the source code, and usually less than 500 bytes in the binary. Human nature is unlikely to be subject to as much change as the architectural characteristics, and the constraints of ease of implementation and manageability are going to continue to restrict the function size.

There have been several efforts to statically reorder functions to optimize programs. In the past five years there has also been a significant effort to dynamically optimize programs at levels of granularity finer than that of a page, although not at the level of functions. In last month's Solid State Circuits Conference Intel reported prototyping portions of a microprocessor operating an order of magnitude faster than the current day giga-hertz processors, showing that the chip speed is doubling more frequently than 18 months [1]. Due to the increasing gap between the processor speed and memory bandwidth, we believe that the techniques to dynamically relocate functions would pay-off.

We have built a simulation platform to non-intrusively find the sequence in which various functions are used during program execution. In this paper we reformulate the Working Set framework in the context of functions, to define and develop relationships between various metrics. We measured these metrics by running some commonly used applications on our simulation platform. Our results indicate that there is a high degree of temporal locality at the level of functions.

2 Related Work

This paper, along with another paper on fixed cardinality working sets, reports on our study of locality at the level of functions. To our knowledge this is the first study of its kind, in which the references to functions are studied. The immediate consequence of temporal locality at the level of functions is to explore the possibility of dynamically optimizing the execution of the *hot* functions - perhaps by packing them together to reduce the working set size and cache conflicts. There has been prior work in changing code layout at the function level at compile time as well as dynamically. There have also been efforts to exploit program locality dynamically at other levels of granularity.

Pettis and Hanson [20], Scott McFarling [18], Hatfield and Gerald [11], Gloy and Smith [15],have presented

methods to rearrange the procedures, which comprise the executable, based on profile data to improve memory locality. Most of these use profile data in the form of a *weighted call graph* (WCG). In a WCG, there is a node for each function and an undirected edge connects two nodes U and V if U calls V or vice-versa. An edge between two nodes has a weight equal to the number of times the functions, represented by the two nodes, call each other. If two functions are placed adjacent in memory, they are less likely to incur conflict misses because they would get mapped to different portions of the cache, unless the sum of their sizes is greater than the cache size. The technique used by Pettis and Hanson, iterates over the WCG in the order of decreasing edge weight. For each edge, it tries to place the functions of the edge as close as possible. Unlike the earlier approaches, Gloy and Smith [15] record temporal ordering information during the profiling stage to get better layouts. These efforts indicate that it is desirable to have a layout in which functions that tend to get used together in a phase of execution, are located contiguously.

The effectiveness of static code layout techniques depends on how well the initial execution, used for gathering the profile data, reflects the nature of program execution resulting from the different data sets used in practice. Modern applications use Dynamically loaded libraries (DLLs), on the Windows platforms, and shared objects, on the unix platforms. Application software often gets shipped as a collection of DLLs. Dynamic linking imposes limits on compile time code layout strategies. Dynamic code generation environments, like Java JITs, also make static code layout techniques impractical. Unlike the static techniques, the cost of restructuring the code does not get amortized over several executions in the dynamic techniques. Dynamic techniques are, therefore, faced with the challenge of recouping the overhead during each program execution.

Chen and Leupen [5] presented a novel heuristic, called *just in time code layout*, which copies functions into memory, when they are first invoked. Thus the order in which the functions appear in the executing image gets fixed when they get loaded into memory. They reported a reduction in the footprint of the executable by 50%. They also showed that their technique provides improvements in instruction misses which are comparable to those achieved by Pettis and Hansen [20].

Bershad et al. [4] reported a dynamic technique to avoid conflict misses in a direct-mapped cache which is much larger than the page size. Their technique works at the granularity of pages and uses a special

hardware device, called the *Cache Miss Lookaside Buffer (CML) buffer*, to detect conflicts. Cache misses for each page get recorded in the CML buffer. When the misses exceed a certain threshold the CML buffer interrupts the CPU. The operating system, then, looks for two or more pages which share the same "cache-page" (portion of the direct-mapped cache to which a page in physical memory is mapped to) and have cache misses higher than a certain threshold. All but one of those pages are copied so that they no longer share the same cache-page.

More recently there have been several efforts involving code transformation during run time, although not at the level of functions, for different purposes. There are systems, referred to as the *dynamic binary translation systems*, which run binaries compiled for one platform on a totally different platform. Crusoe [17] achieves power efficiency by translating the x86 code presented to it and running it on a power efficient VLIW processor which is built with fewer transistors. DAISY [13] achieves high performance by translating *PowerPC* binaries for the DAISY VLIW processor. Shade [7] and Embra [25] provide faster simulation capabilities. Strata [23] is an infrastructure for software binary translation, which can be extended to build translation systems for implementing specific functionality. The key factor in all of these technologies is the judicious choice of sufficiently frequently executed pieces of code for translation into native code, while interpreting the less frequently used portions of the program.

There are also dynamic optimization systems which exploit locality in code to aggressively optimize code at run time. Dynamo [2] and Mojo [6] are both user level software efforts to optimize programs executing on the HPUX/PA-RISC and Windows/x86 platforms, respectively. Replay [19] is a new processor framework that supports dynamic optimization, in hardware. Kistler and Franz [16] presented a dynamic optimization system by utilizing the idle time and dynamically collecting execution profiles.

The above dynamic optimization and translation systems work by selecting pieces of code at the *basic block* (a maximal sequence of machine instructions which can be entered only at the beginning and exited only at the end) level, and often chaining blocks which get executed one after another at run time to identify *hot* paths. The selected portion of the code is then translated and/or optimized with the expectation that the overhead involved would be more than made up for, by the repeated execution of the optimized code. The success of such research efforts indicates temporal locality at the block level. However, temporal locality

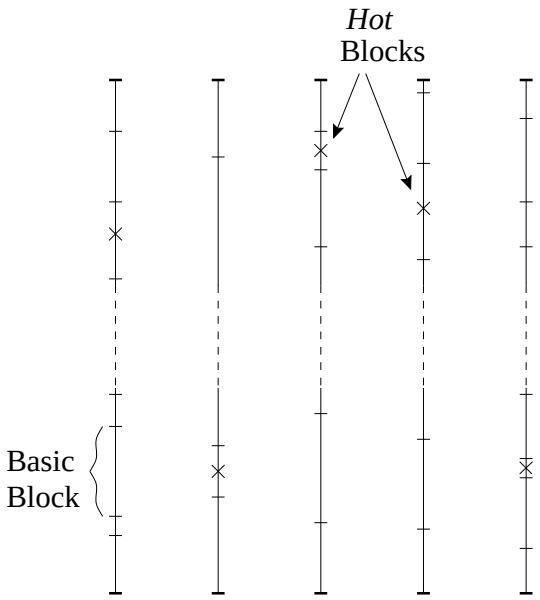

Figure 1: Above figure depicts a program with 5 functions. The *hot* blocks are marked with an 'X'. While a footprint of basic blocks would contain only a portion of the code, a footprint of functions would contain the entire code. The block level locality does not translate into a function level locality.

at the block level does not necessarily imply temporal locality at the function level.

Consider an example, as shown in Fig 1, in which the hot blocks are spread across all the functions of the program. If we assume that each function, on an average, has 10 blocks and the hot blocks are the only ones which get accessed then only one-tenth of all the blocks would be in a footprint consisting of blocks. On the other hand, a footprint consisting of functions would require all the functions, consuming much more space. This example demonstrates that locality at the block level does not automatically imply locality at the function level.

The above techniques for dynamically optimizing code involve disassembling the binary executable followed by aggressive optimization of code. Because of the high overhead, they have to be fairly selective to ensure that the code which is optimized would in fact be repeatedly executed in the future. There have been cases for which Dynamo and Morph had to "bail out", i.e., stop optimizing the binary and just execute it normally. We believe that simpler heuristics for function re-ordering, having a lesser overhead, would be less susceptible to such failures. Moreover, function level optimization does not necessarily rule out the

optimization techniques used in the above dynamic optimization systems completely. It is possible to use both approaches simultaneously.

The success of schemes which dynamically re-order functions hinges on the existence of a working sets at the function level and methods for identifying them accurately and efficiently. In this paper we report on the experiments which show that programs exhibit a working set nature at the function level. While there is a large body of work on program behavior [24, 14, 9, 3, 21], to the best of our knowledge, there has been no effort to study locality at the level of functions.

3 Background

3.1 Definitions

Let $N = \{1, 2, \ldots n\}$ be the set of functions which make up the program. Its dynamic behavior can be modelled as a sequence of accesses to the functions. Let $\rho = f_1, f_2, \ldots, f_k$ represent the sequence of function accesses, where $f_t \in N$ is the function referenced at *virtual time* t. As different instructions in the function get accessed, that function does not get successively repeated in the sequence. There is only one entry in the sequence corresponding to all the accesses which happen in the interval of time from the instant when the control enters the function to the instant when the control leaves the function. Thus a return or call to a different function would result in a new element in the sequence. Note that an alternative formulation where the sequence grows for every access to a machine instruction exists. We do not choose that one mainly because we want to study the locality in the usage of different functions and, therefore, want to eliminate the affects of code locality within a function. Thus the virtual time does not "progress" uniformly with respect to the real time.

We define the program's working set of functions $W(t, T)$ at virtual time t to be the set of distinct functions in the T most recent references $f_t, f_{t-1}, \ldots, f_{t-T+1}$. In the special case where the program has just begun execution and $T > t > 0$, $W(t, T)$ consists of all the distinct functions in $f_t, f_{t-1}, \ldots, f_1$. When $t = 0$, $W(t, T) = \emptyset$. Let $w(t, T)$ be the size of the working set of functions at time t, i.e., $w(t, T) = |W(t, T)|$. The *average working set size* of functions, $w(T)$ has the obvious definition: $w(T) = \frac{1}{k} \sum_{t=1}^{k} w(t, T)$. Every

time a function, which is not already in the working set, is accessed there is a "miss". The miss rate $m(T)$ is defined to be the number of misses divided by the length of the sequence when the window size is T. If T is chosen to be 1, then every access in ρ should result in a miss. We also define a function interreference frequency function, called g. $g(x)$ is the empirical probability that the virtual time interval between successive references to the same function is x. Similar metrics were defined by Denning and Shwartz in [10]; the differences are in the way the reference string is defined.

3.2 Relationships between $w(T)$, $m(T)$ and $g(x)$

The metrics $w(T)$, $m(T)$ and $g(x)$ are inter-related as shown below:

$$m(T) = w(T+1) - w(T) + \frac{w(k,T)}{k} \quad (1)$$

$$g(T) = m(T-1) - m(T) \quad (2)$$

Similar relationships were developed earlier in the terms of page references in [12] and [24, pages 56–60]. We briefly describe the proof here. Let the *function miss characteristic function* χ be defined as follows:

$$\chi(t,T) = \begin{cases} 1 & \text{if } f_t \notin W(t-1,T); \\ 0 & \text{otherwise.} \end{cases}$$

χ tells us whether there is a "function miss" at a particular instant of time. It is naturally related to $m(T)$ as $m(T) = \frac{1}{k}\sum_{t=1}^{k}\chi(t,T)$ and it can be determined from w as well: $\chi(t,T) = w(t,T+1) - w(t-1,T)$, where $0 < t \leq k$. Substitution of the value of χ from the latter equation into the former, followed by some algebra, results in equation(1).

Let $\zeta(t)$ be the interreference distance associated with the reference at time t, f_t. In the case when f_t does not get accessed earlier $\zeta(t)$ is defined to be ∞. Thus when $\zeta(t)$ is finite, $f_{t-\zeta(t)} = f_t$ and none of

$f_{t-\zeta(t)+1}, \cdots, f_{t-1}$ are same as f_t. It follows that,

$$\begin{aligned}
\chi(t,T) &= 1 \text{ iff } \zeta(t) > T \\
\Rightarrow \frac{1}{k}\sum_{t=1}^{k}\chi(t,T) &= \text{Rel. Freq. of } (\chi(t) > T) \text{ in } \rho \quad \text{OR} \quad \text{Freq}[\zeta(t) > T] \\
g(T) &= \text{Freq}[\zeta(t) = T] \\
&= \text{Freq}[\zeta(t) > T-1] - \text{Freq}[\zeta(t) > T] \\
&= m(T-1) - m(T)
\end{aligned}$$

3.3 Measurement of $w(T)$, $m(T)$ and $g(x)$ from the Reference String

Given a function reference string generator, an efficient "on-line" technique, which does not require the storage of the references, exists. This technique is based on equations(1) and (2) and a detailed description can be found in [24, Pages 146–147]. We have presented the pseudo code in figure 2. On completion, the arrays MeanWSSize, FaultRate and g contain the values for $w(*)$, $m(*)$ and $g(*)$, respectively, for all values up to a predefined constant TMAX. The on-line nature of this technique makes it particularly attractive to us because reference strings can get very long.

4 Simulation Methodology

Our simulation platform is composed of the following subcomponents: the Shade [8] library, Function Information Extractor, Function Store, and the Function Reference Processor. Fig 3 shows the interactions between them. At the top level our simulation platform is a loop in which each iteration simulates one instruction of the program. It checks the program counter(pc) to see whether the pc still points to a location in the function most recently accessed. If that is not the case the new instruction represents a new reference at the level of functions and it is used in the estimation of the three metrics – average working set size, miss rate and interreference frequency. All of these actions are initiated by the component labeled *Main* in Fig 3.

The component labeled Function Information Extractor provides functionality to obtain the code layout (sizes of the functions and their virtual addresses) from the symbol table in the application binary. This information is stored in a data structure, provided by the module labeled Function Store, which can be queried

```
for each function, x, in the program  // Initialization Phase..
      FuncRefTime[x] = -TMAX
for each value, i, from 1 to TMAX
      g[i] = adj[i] = 0
Time = 0

for each function reference, f, in the reference string  // On-line Phase..
      Time = Time + 1
      delta = Time - FuncRefTime[f]
      if delta < TMAX
               g[delta] = g[delta] + 1
      else
               g[TMAX] = g[TMAX] + 1
      FuncRefTime[f] = Time

RefStringLength = Time  // Post execution phase..
for i = 0 to n-1
      delta = RefStringLength + 1 - FuncRefTime[i]
      if delta < TMAX
               adj[delta] = 1/RefStringLength
MeanWSSize[0] = 0; FaultRate[0] = 1; f1 = 1;
for i = 1 to TMAX
     g[i] = g[i] / K
     FaultRate[i] = FaultRate[i-1] - g[i]
     MeanWSSize[i] = MeanWSSize[i-1] + f1
     f1 = f1 - g[i] - adj[i]
```

Figure 2: Technique for measuring the metrics w, m and g

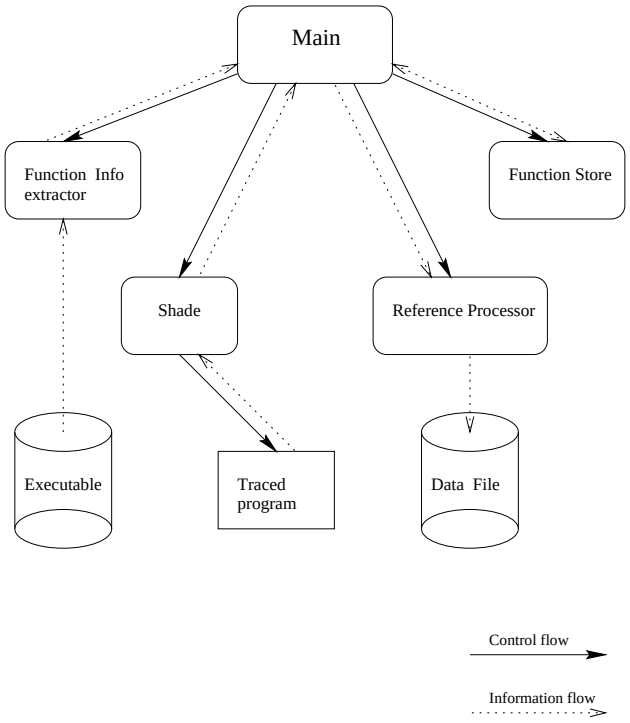

Figure 3: Simulation Methodology

efficiently during the application's (simulated) execution. Shade is a powerful library of routines which provides the support to identify the application to be simulated, load the application into the simulated run time environment, "run" the application and monitor the pc constantly. The above 3 components, together, provide sufficient functionality to obtain the sequence of function references, which is continually fed into the Function Reference Processor to measure the necessary metrics. Thus, the Function Reference Processor implements the techniques to measure the metrics which have been described in detail in the previous section.

5 Workloads

For this study we selected applications which were not interactive in nature to ensure that our experiments are reproducible. Other factors which influenced the choice of workloads were ease of availability and the number of functions the executable contained. Since we are trying to show that working sets are smaller than the whole program, we avoided programs with fewer than 100 functions.

The programs we studied were awk, unzip, troff, perl and dvips. We use awk to run an awk script which

finds C printf statements. The input to the script was an older version of a C source file used for building the simulator. We used unzip to extract files from an archive of about 31 kilobytes containing 27 text files. The troff workload invokes troff with a special command line option to strip off all the troff formatting commands from a man page on "ct", a communication command on Solaris. Perl is used to interpret a perl script which finds out user id, group id and supplementary group ids. Dvips is used to convert a 12 page postscript draft consisting of text, figures, tables and references into a dvi file.

6 Observations

In this section we present results which show that

- There is a high degree of correlation between the current function references and those in the immediate future, while those farther away in time tend to be more or less uncorrelated.
- Variable size working sets of functions based on a working set parameter almost always perform better than the fixed sized working sets based on an LRU replacement policy, in terms of the average number of functions maintained in the working set.

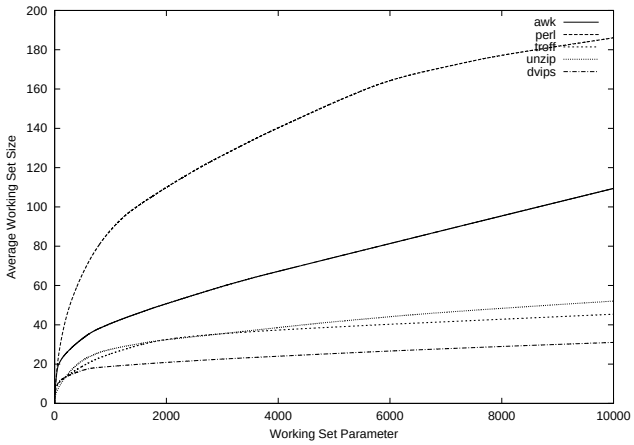

Figure 4: Working Set Curves.

In figure 4 we have presented the working set curves for all the workloads. The average working set size during the complete execution of the program is plotted against the different working set parameters, up to 10000, used for determining the working set. All curves are convex (or convex up). This implies that

increasing the working set parameter (or the window into the most recent references) results in a less than proportionate increase in the working set size. In other words, if the working set parameter is increased from x to $2x$, the new working set size, $w(2x)$, is lesser than $2w(x)$. This indicates that there is temporal locality in the references to functions – the "current references" (in the window of size x) also appear in the vicinity (of window size $2x$) resulting a less than proportionate increase (of $w(x)$).

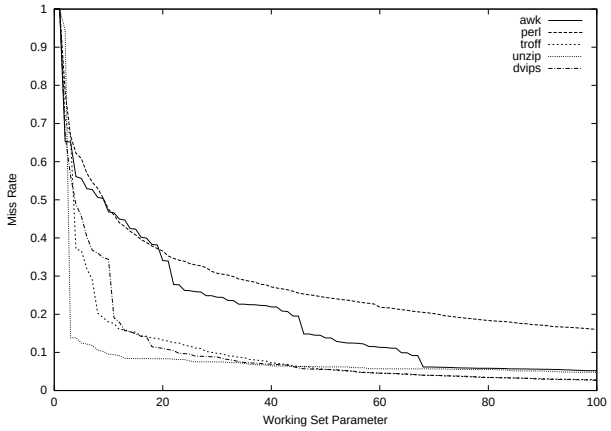

Figure 5: Miss Rate Curves.

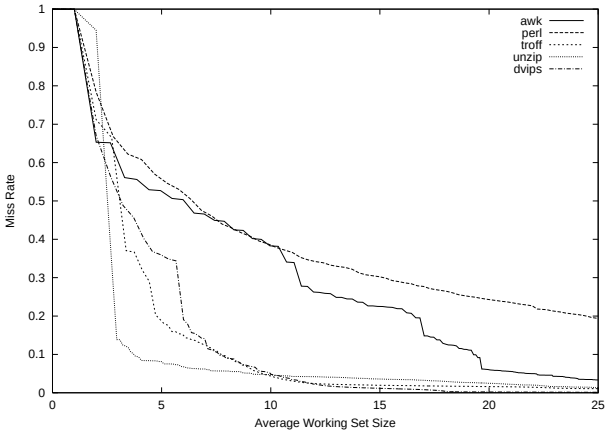

Figure 6: Miss Rate Versus Average Working Set Size.

Figure 5 is a plot of the miss rates against different values of the corresponding working set parameters. Note that a "miss" is a reference to a function which is not already in the working set. While we measured the miss rate for working set parameters up to 10000, for most of the larger values the miss rate is close to 0. Thus a miss rate plot for all measured values of working set parameter, for all workloads, results in an "L" shaped curve with the arms of "L" lying undiscernibly close to the Y and X axes. We have, therefore, chosen to provide a plot for smaller values on the X axes in Fig. 5. The average working set size is a somewhat crude estimate of the actual memory space occupied. Nonetheless, the variation of the miss rate with the amount of memory used, is an interesting aspect. Figure 6 presents this trade-off.

Figure 7 shows the distribution of the function interrefence distances for each workload separately. These graphs clearly show that the relative frequency of larger interreferences distances is close to zero, while the frequency of smaller interreference distances is very high. Thus, current function references form a very good indicator for references in the immediate future, while there is not much correlation between the current function references and those in the distant future. An interesting feature in all these graphs is the random

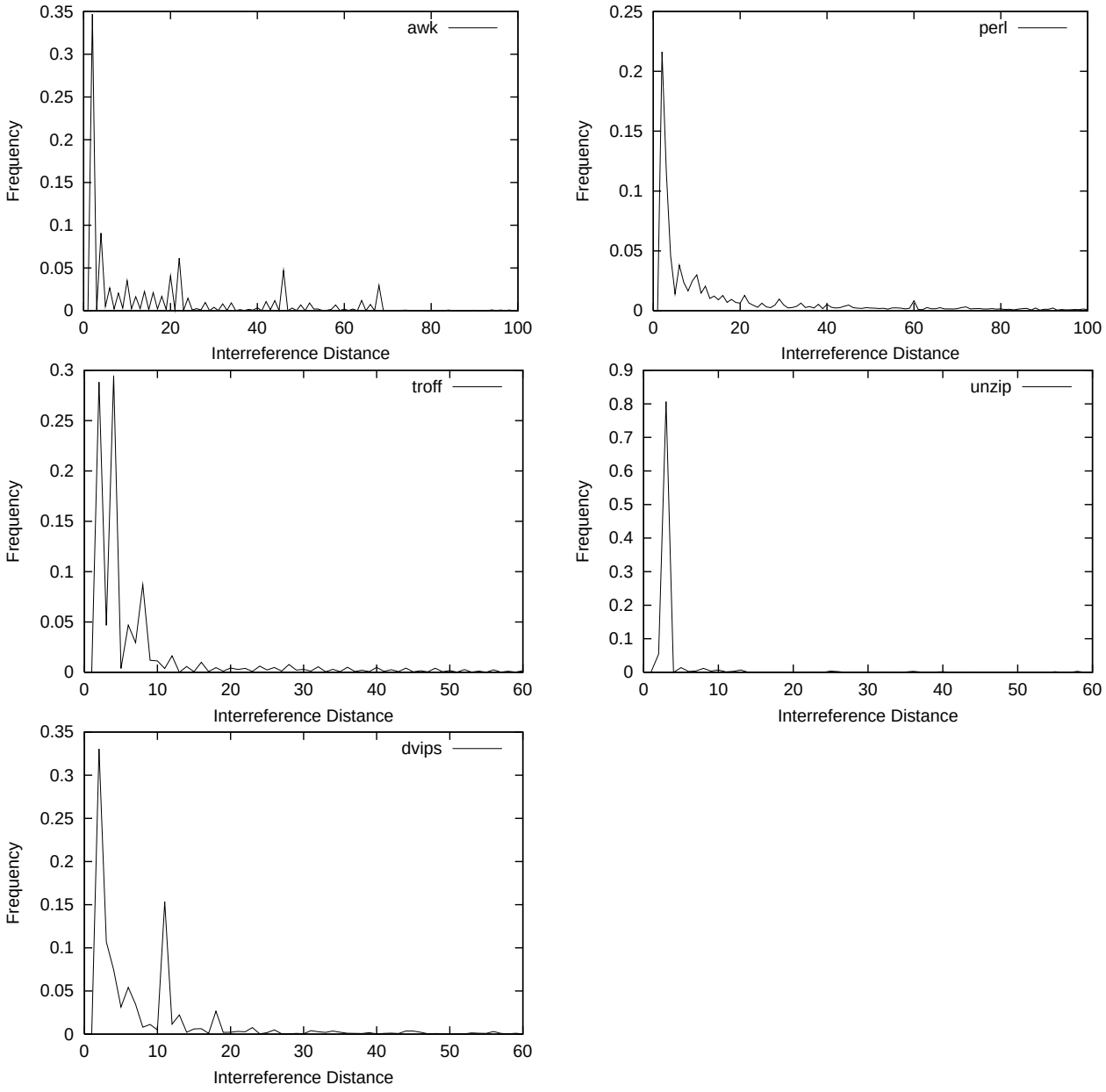

Figure 7: Interreference frequencies for all workloads

fluctuation of the relative frequency (instead of a smooth curve) as it drops towards the X axis. Similar behavior was observed in references studied at the page level [24, pages 163–164].

Workload	Av. WS size for 2% miss rate	% size
Awk	16.64	4.5
Perl	137.65	11.56
Dvips	12.29	4.7
Troff	14.58	4.9
Unzip	22.02	18.97

Table 1: Shows the average working set size which results in a 2% miss rate. Column 2 gives the absolute value while column 3 gives it as a percentage of the total number of functions in the program

In an earlier paper, which we submitted for publication, we studied these workloads with constant cardinality working sets maintained by an LRU replacement policy. We observed that it is necessary to have a working set size consisting of 40% of the program's functions to ensure a miss rate less than 2%. In fact in all cases, except unzip, we found that a 20% fixed size working set was sufficient for a miss rate less than 2%. In table 1 we present the average working set sizes which result in a 2% miss rate when a variable size working set (based on a working set parameter) is used. We observe that a working set which is 20% of the program size is more than enough to ensure a miss rate of less than 2%, for all workloads, in the case of variable size working sets. In this case also, it is true that a larger working set is necessary only for the unzip workload – for all the other workloads a working set which is half as large (around 10%) is sufficient to ensure a miss rate below 2%. Table 2 further illustrates the fact that the variable size working set almost always performs better than the constant size LRU based counterpart. Some of the data in this table has been reproduced from our earlier work on constant cardinality working sets.

The program footprint size, in terms of number of functions, changes with time during program execution. The possible explanation for better performance of variable size working sets is that, when the footprint is smaller, the working set parameter based technique caches fewer functions with perhaps precisely those

Workload	WS = 10% of Pgm Size		WS = 15% of Pgm Size	
	fixed	variable	fixed	variable
Awk	2.11	1.90	1.37	1.13
Perl	2.70	2.16	1.80	1.78
Dvips	0.27	0.14	0.14	0.07
Troff	0.84	0.89	0.23	0.23
Unzip	6.90	4.26	5.89	3.25

Table 2: Shows the miss rates for 2 different kinds of working sets: constant cardinality working sets maintained using LRU replacement policy and traditional (variable size) working sets based on a window into the most recent references.

functions which are in the footprint. This results in smaller average working set sizes.

7 Exploiting Locality at the Function Level

Traditional harvard architectures, in which there is a separate instruction cache and data cache, do not leverage the locality at the level of functions. The cache line is a fixed power of two while function sizes are variable and often larger than a single cache line. More importantly, hot (or frequently used) functions tend to be spread out in the physical address space resulting in cache conflicts within themselves. This prevents the instruction cache from capturing all the hot functions at the same time. The fact that conflicts do occur in hot functions in practice, is supported by the improvements witnessed by the compile time techniques to re-order the function layout. These techniques improved the performance by laying out functions (which were presumed to be hot and likely to be used around the same time during execution) contiguously – thus reducing the chances that there would be cache conflicts within them. Another advantage of caching hot functions is the possibility of reducing the working set of pages and the demand for the limited tlb entries.

In this section we outline the basic features and changes necessary to build a platform (hardware, operating system and code generation tools) which would be capable of caching functions by dynamic function

relocation. Position independent code, where the transfer of control is specified as a displacement from current address, would be necessary to ensure proper control flow within a function in the presence of movement of functions during run-time. Dynamically linked libraries currently utilize this kind of code in order to cope with the lack of knowledge of the absolute location of the dynamically linked function in the virtual address space before run time[1].

Another aspect which needs to be addressed is the proper transfer of control during function *call*s and *return*s. In our discussion a *call* refers to one or more instructions which get executed from the point at which the caller decides to invoke the callee until the point at which the control enters the callee. It does not refer to a specific instruction in the instruction set architecture of any processor. Similarly, a *return* refers to one or more instructions which get executed from the point at which the callee decides to end its invocation until the point at which the control enters the caller. The main changes in the function call and return are as follows:–

- Before transferring control to the callee, the callee's address would have to be fetched and then the control should be transferred to the callee. Instead of saving the return address on the stack, the offset of the return location from the beginning of the caller and the identity of the caller would have to be saved on the stack.

- The corresponding change in the return is that the return address would be found out by adding the current location of the caller and the offset and then control would be transferred to the return address. As usual, the "return address" (offset and caller id in our case) is popped from the stack.

Finally, there has to be an agent which copies the hot functions to a faster memory, or perhaps just contiguous locations in order to reduce conflicts between hot functions and the number of pages in the working set (thus reducing tlb misses). This agent would also update the current address of the function, which it relocated, in a *well known area* used for keeping track of all the relocatable functions. Note that the movement of the function and the corresponding update in the well known area have to appear atomic in nature to the process to which the function belongs. Actual implementation of this agent could take the

[1] Dynamic linking typically postpones the placement of the function in virtual address space to run-time. It does not achieve transparent relocation of functions during run-time

form of a daemon, a kernel process which gets scheduled periodically. The relocations themselves can be scheduled when the cpu is in the idle loop, thus hiding the overhead of the relocation at least to some extent. The well known area for maintaining function locations could be implemented by a per process table which appears in each process's address space and gets initialized by the compiler and loader to contain the original function locations.

Given the temporal locality in the functions, the relocation agent can predict the functions which would be used in the near future effectively. If necessary per function usage statistics can be maintained in the well known area for each process.

We note that such a change, to implement dynamic relocation of functions, could also have architectural implications since it could result in a change in the distribution of instructions in the instruction set architecture, especially making those in the call and return sequence more frequent. Improving the speed of those machine instructions and making available special purpose registers for them, could have an impact on the overall system performance.

8 Conclusions

In this paper we have presented the results of our study of program behavior at the level of functions using the traditional concept of working sets developed during the study of paging. In another paper, which is currently under review, we have presented the results of maintaining constant cardinality working sets. To the best of our knowledge, this is the first study of its kind, in which the program behavior has been studied at the level of functions. We have shown that there is a substantial amount of locality at the level of functions – we observed that caching about 10% of the functions is usually enough to ensure a miss rate of less than 2%. We have also shown that variable size working sets are better than fixed cardinality working sets, indicating that the function footprint of a program tends to change during execution. Our results clearly show that it is feasible to predict function references in the immediate future with a high degree of accuracy.

Note that, unlike the earlier work on paging in which every memory reference resulted in a corresponding reference at page level, our formulation of the working set framework results in new function level references only when the next memory reference is to a function which is different from the current one. This has the

effect of suppressing intra function locality and giving a better estimate of the inter function locality.

While we constrained ourselves to use non-interactive and reproducible workloads, we believe that workloads like editors, browsers and windowing software would have inherently more temporal locality than the workloads we selected. Such software is rich in functionality and typically only a few features are utilized repeatedly, resulting in only a small portion of the code being exercised many times. Software used in the real world is, therefore, likely to be more amenable to the exploitation of temporal locality at the function level.

We have shown that it is feasible to build systems which can dynamically relocate functions. Various policies for relocation of functions have to be evaluated to see whether the cost of relocation is offset by the advantage obtained by avoiding cache conflicts and tlb misses. If not already attractive, function relocation which is more cpu intensive and less memory intensive than misses in the memory subsystems, is going to become a viable option with the constant rise in processor speeds. We are currently doing a complete machine simulation using SimOS [22] to study this trade-off.

References

[1] M. Anders, S. K. Mathew, B. Bloechel, R. Krishnamurthy, K. Soumyanath, and S. Borkar. A 6.5ghz 130nm single-ended dynamic alu and instruction-scheduler loop. In *IEEE International Solid-State Circuits Conference*, Feb 2002.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[3] Thomas Ball and James R. Larus. Using paths to measure, explain and enhance program behavior. *IEEE Computer*, 33(7):57–65, July 2000.

[4] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170. ACM, 1994.

[5] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with Just-In-Time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32. USENIX Association, August 1997.

[6] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[7] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[8] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Laboratories, Inc. and University of Washington, 1993.

[9] Peter J. Denning. The working set model for program behavior. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 323–333. ACM, November 1967.

[10] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. In *Communications of the ACM*, volume 15, pages 191–198. ACM, 1972.

[11] D.J.Hatfield and J.Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[12] D.R.Slutz and I.L.Traiger. A note on the calculation of average working set size. *Communications of the ACM*, 17(10):563–565, Oct. 1974.

[13] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transaction on Computers*, 50(6):529–548, June 2001.

[14] Domenico Ferrari, editor. *The Measurement of Program Behavior*, volume 9 of *Computer*. IEEE Computer Society, Nov 1976.

[15] Nikolas Gloy and Michael D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[16] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transaction on Computers*, 50(6):549–566, June 2001.

[17] Alexander Klaiber. The technology behind crusoe(tm) processors. Transmeta White Paper, January 2000.

[18] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[19] Sanjay J. Patel and Steven S. Lumetta. rePLay: a hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[20] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] Vidyadhar Phalke. *Modeling and Managing Program References in a Memory Hierarchy.* PhD thesis, Rutgers, The State University of New Jersey, October 1995.

[22] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, 1995.

[23] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the IEEE 2001 Workshop on Binary Translation*, September 2001.

[24] Jeffrey R. Spirn. *Program Behavior: Models and Measurements.* Operating and Programming Systems Series. Elsevier North-Holland, Inc., 1977.

[25] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS conference on measurement and modeling of computer systems*, pages 68–79, 1996.